CBC IMPORT·EXPORT

AMSTRAD MASKINKODE

Søren Grynnerup

AMSTRAD MASKINKODE

Søren Grynnerup

Forlag: Copenhagen Book Centre ApS
Roskildevej 338
2630 Tåstrup
(02) 99 17 99

Copyright: (c) 1985 Søren Grynnerup

1. udgave

ISBN 87-88739-02-3

Mekanisk, fotografisk eller enhver anden gengivelse af denne bog eller dele af den er ikke tilladt uden skriftlig accept fra forlag og forfatter.

INDHOLDS-FORTEGNELSE

1. Hvad er maskinkode.....5
2. Registre.....10
   A-registret.....13
   BC-registrene.....19
   DE-registrene.....20
   HL-registrene.....21
   Indeksregistrene.....22
   F-registret.....23
   Aftryksregistrene.....34
   R og I registrene.....36
   PC-registret.....37
3. Lagring af konstanter i registre.....38
   Lagring i 8-bit registre.....38
   Lagring i 16-bit registre.....43
4. Instruktioner til indirekte registerlagring.....46
   Lagring mellem 8-bit registre.....47
   Lagring mellem 16-bit registre.....54
5. Lagring mellem registre og AMSTRADs lager.....56
   Lagring af registre med indhold af lageradresser...57
      Direkte lagring 58
      Indirekte lagring.....60
      Indekseret lagring..... 63
6. Ombytnings-instruktionen.....66
7. Stakpeger-instruktionerne.....69
8. Aritmetik instruktionerne.....78
   Additions-instruktionerne.....79
      ADD instruktionen.....79
      ADC instruktionen.....89
      INC instruktionen.....94
   Subtraktions-instruktionerne.....98
      SUB instruktionen.....99
      SBC instruktionen.....103
      DEC instruktionen.....105
9. Sammenligningsinstruktionerne.....111
   Bloksøgnings instruktionerne.....114
10. Logiske instruktioner.....117
   AND instruktionen.....118
   OR instruktionen.....124
   XOR instruktionen.....126
11. JUMP-instruktionen.....128
12. DJNZ-instruktionen.....135
13. CALL-instruktionen.....143
14. RET-instruktionen.....149
15. Rotations/Skift-instruktionerne.....154
   Multiplikation og division.....164
16. Enkelt bit instruktionerne.....170
   BIT instruktionen.....171
   SET instruktionen.....173
   RES instruktionen.....175
17. Bloklagrings instruktionerne.....176

18. Restart instruktionerne.....179
RST 0.....180
RST 8.....180
RST 16.....183
RST 24.....184
RST 32..... 185
RST 40.....186
RST 48.....186
RST 56.....187
19. Interrupt instruktionerne.....189
20. Input/Output instruktionerne.....196
21. Restgruppen.....200
22. Nyttige adresser.....203
23. Appendix.....207
Assembler kode sorteret alfabetisk.....207
Decimal kode sorteret kronologisk.....217
Talsystemer.....224
Indeksregister instruktioner.....229
Flageffekt.....230
Oversigt over operationstider.....234

HVAD ER MASKINKODE ?

Maskinkode er et programmeringssprog nøjagtig som BASIC er det. Man kan altså skrive programmer i maskinkode svarende til BASIC-programmer. Der er imidlertid væsentlige forskelle mellem de to 'sprog'. Den afgørende forskel er at mikroprocessoren, der sidder i din computer kun "forstår" og derfor kun kan udføre instruktioner, som den modtager i maskinkodesprog.

Når din computer alligevel forstår BASIC-sproget hænger det sammen med at AMSTRAD-folkene har lagt maskinkode instruktioner i maskinen, der skal udføres hver gang et bestemt ord (BASIC-kommando) indtastes eller mødes i udførelsen af et program. Samlingen af den maskinkode som ligger i AMSTRAD til oversættelse af BASIC-sproget kaldes naturligt nok for FORTOLKEREN. Denne er skrevet i maskinkode for at opnå maksimal hastighed i udførelsen af dine BASIC-kommandoer.

Hermed har vi allerede en af de væsentligste årsager til overhovedet at læse sig programmering i maskinkode. Der er imidlertid knyttet flere såvel fordele som ulemper til maskinkodeprogrammering.

Fordelene består hovedsaglig i

- at programmerne udføres væsentlig hurtigere end tilsvarende BASIC programmer.

- programmeringen giver mulighed for at lave rutiner det ikke var muligt at lave med BASIC.

- programmerne udnytter AMSTRADs lager bedre, idet rutinerne kun indeholder de absolut nødvendige rutiner. BASIC-fortolkeren skal være så fleksibel at den kan bruge samme kommando til mange forskellige formål i modsætning til dine maskinkodeprogrammer.

- maskinkode programmer kan gøre dig uafhængig af AMSTRADs indbyggede operativsystem og BASIC-fortolker. Du kan således koble disse ud og bruge samtlige 64 Kilobyte til egne programmer.

Maskinkode programmer udføres meget hurtigere end de tilsvarende BASIC-programmer, fordi BASIC-kommandoerne først skal oversættes til maskinkode for at kunne udføres. Egentlig sker der ikke en reel oversættelse svarende til en kompilering, men de indtastede koder kalder i udførelsesøjeblikket en maskinkoderutine i maskinens lager. Da disse rutiner ofte er indrettet til at kunne klare flere varianter af en given kommandoer opstår noget man kalder -overhead-. Overhead er den del af den kaldte rutine, som man ikke behøver at lade computeren udføre, men som, af hensyn til variationen indenfor kommandoens anvendelse, skal være tilrådighed. Eksempelvis skal vi senere i bogen se hvordan vi får udskrevet tekst på Monitoren. Dette sker ved kald af en udskrivningsrutine i AMSTRADs lager. Rutinen kaldes imidlertid ikke på samme sted som AMSTRAD ville gøre det i forbindelse med en PRINT-kommando eksempelvis. Vi hopper i stedet ned midt i rutinen, nemlig lige til den del af rutinen vi netop har brug for. Herved hopper vi uden om forskellige tests som systemet normalt ville udføre på det der skal udprintes. Således skal AMSTRAD først finde ud af om printet skal sendes til printeren eller skærmen. Herefter skal testes om det udskrevne er i ASCII-kode, om det er et linienummer osv. Alt dette har vi muligheden for at hoppe uden om, idet vi på forhånd ved hvad vi vil udskrive - hvorfor vi således ikke har brug for de mange forudgående (tidskrævende) tests.

Endvidere skal AMSTRAD forud for udførelsen af maskinkoden først teste om den indtastede BASIC-kommando er defineret. Denne test tager naturligvis også tid.

Når du lærer at programmere i maskinkode vil du samtidig lære de metoder din computer bruger eksempelvis til at lagre BASIC-programmer på etc. Skal du udføre en opgave der har forbindelse hermed kan du naturligvis 'skræddersy' dine programmer til netop denne opgave. Dette er væsentlig vanskeligere når man skal lave et operativsystem. Her må alt gøres så 'generelt' som muligt for at kunne tilgode se flest brugere.

Maskinkode programmer fylder mindre i lageret end de tilsvarende BASIC-programmer. Dette hænger igen sammen med at du 'nøjes' med de instruktioner, der netop løser din opgave - hverken mere eller mindre. Dit program bliver herved ikke særligt fleksibelt, hvilket imidlertid ikke har nogen betydning.

Endelig kan maskinkode programmering gøre dig uafhængig af AMSTRADs operativsystem og BASIC-fortolker. Disse to basis 'programmer' indeholder jo alle de spilleregler du må indordne dig under for at programmere i BASIC. Når kan programmere i maskinkode kan du tilføje eller mindske (!) antallet af BASIC-kommandoer efter behov. Dette lader sig ikke gøre i BASIC alene. Endvidere vil du kunne få udført specielle opgaver som dine BASIC-programmer ikke ville kunne, eksempelvis at få udført to programmer samtidig.

Denne uafhængighed medfører imidlertid ikke kun fordele. Den fejlrapportering, der stilles til rådighed ved programmering i BASIC er du udelukket fra at udnytte, når du begynder dine egne maskinkodeprogrammer. 'Crasher' maskinen, d.v.s. reagerer den ikke på dine indtastninger etc, er der ingen vej tilbage - der er kun at slukke for maskinen, tænde igen og så indlæse programmet igen. I denne forbindelse er det derfor MEGET VIGTIGT at du tager en sikkerhed kopi af dine (længere) programmer. Da AMSTRAD har cassetten indbygget er det næsten for tåbeligt at glemme dette. Ellers risikerer du naturligvis at skulle indtaste hele programmet igen efter et crash.

Hermed er vi allerede godt på vej til at tale om ulemperne ved maskinkodeprogrammering. Disse kan groft set siges at være:

- programmerne vil være om ikke umulige at 'læse' så uhyre meget vanskeligere end BASIC-programmer. En udenforstående vil derfor behøv meget lang tid for at kunne læse og forstå funktionen i blot middelstore maskinkodeprogrammer.

- fejlfinding, som nævnt, er langt mere kompliceret, da man hverken har fejlmelding eller noget editorsystem.

- antallet af instruktioner er væsentlig større end i BASIC-programmer. Dette må være klart, idet BASIC jo består af mange instruktioner i sig selv.

- enhver form for (aritmetisk) funktion er mere kompliceret at programmere. Dette gælder simple additioner,subtraktioner osv., men også strengbehandling etc.

Når dette er sagt er det på tide at sige noget om hvad maskinkode egentlig er og hvordan man opbygger og får udført programmer i dette 'sprog'. Maskinkode er som sagt det eneste 'sprog' som AMSTRADs mikroprocessor forstå. Maskinkode består

alene af binære tal samlet i grupper af 8. Det vil altså sige at hver gang maskinen modtager 8 binære cifre, som hver især kun kan antage værdien 0 eller 1, vil den udføre en ganske bestemt funktion bestemt af VÆRDIen af de 8 binære tal den modtager. Nu er AMSTRADs POKE-kommando indrettet således at når vi POKEr tal ned i lageret, så omsættes disse til binære tal inden de lagres. Hvis vi derfor lagrer en række af tal efter hinanden i lageret, fortæller AMSTRAD hvor de ligger og sætter den igang med at læse dem - ja, så vil mikroprocessoren udføre funktionerne der er bestemt for disse tal. Et simpelt eksempel ville være - i maskinkode - at lagre et bestemt tal i en bestemt lageradresse. Hvis denne adresse eksempelvis er 20000 kan du i det nedenfor viste 'maskinkode program' udskifte 'n' med tallet du vil lagre:

```
62 n 50 32 78 201
```

Når dette program er lagt ned i AMSTRADs lager og udført vil du kunne PEEKe adresse 20000 og se at værdien rent faktisk ligger på adressen.

Skulle du stadig være lidt uforstående overfor hvad der skal ske så udfør det nedenfor viste:

```
POKE 10000,62
POKE 10001,100   (100 indsat for 'n')
POKE 10002,50
POKE 10003,32
POKE 10004,78
POKE 10005,201

POKE 20000,0
CALL 10000
PRINT PEEK (20000)
```

Der er naturligvis intet i vejen for at lægge disse kommandoer ind i et BASIC-program.

Det du nu har gjort er netop at have opstillet dit første maskinkodeprogram og udført det. Maskinkode er altså blot en stribe tal, hvis forskellige værdi - og kombination som vi skal se det - får computeren til at udføre de angivne funktioner. Da de viste tal ikke er til at huske at man opstillet navne for samtlige instruktioner, som mikroprocessoren kan udføre. I vort program fik vi udført instruktionerne:

```
Assembler       Decimal kode

LD A.100        62 100
LD (20000).A    50 32 78
RET             201
```

Første linie svarer til BASIC-kommandoen

```
LET A=100
```

mens anden linie svarer til

```
POKE 20000,A
```

og sidste linie til END, d.v.s. slut på program.

Vi skal senere komme tilbage til hvad dette betyder i detalje. Faktisk beskæftiger resten af bogen sig med gennemgang af disse instruktioner og vi skal se dem i funktion i mange forskellige opgaver.

Når du skal i gang med at lave dine programmer er det i maskinkode programmering lige så vigtigt som i BASIC, at du starter med blyant og papir. Tegn en figur der illustrerer den funktion som du skal have udført. Der findes mange teknikker til tegning af sådanne figurer - rutediagram, datagrammer, Ganthkort etc. Brug disse eller dine egne - START ALTID PÅ PAPIRET.

Når din figur er færdig og du skal til at omsætte den til maskinkode så start med at nedskrive instruktionerne på papir. Der findes flere andre hjælpemidler i denne fase. Disse kaldes alle for Assemblere og kan være en uvurderlig hjælp til opbygning af programmer. Disse Assemblere tillader dig at indtaste instruktionsnavnene i stedet for at POKE talværdier ned i lageret. Du vil derfor hurtigere lære instruktionsnavnene med en Assembler, men indledningsvis kan du klare dig uden. Når du er blevet bare lidt mere erfaren er Assembleren uundværlig.

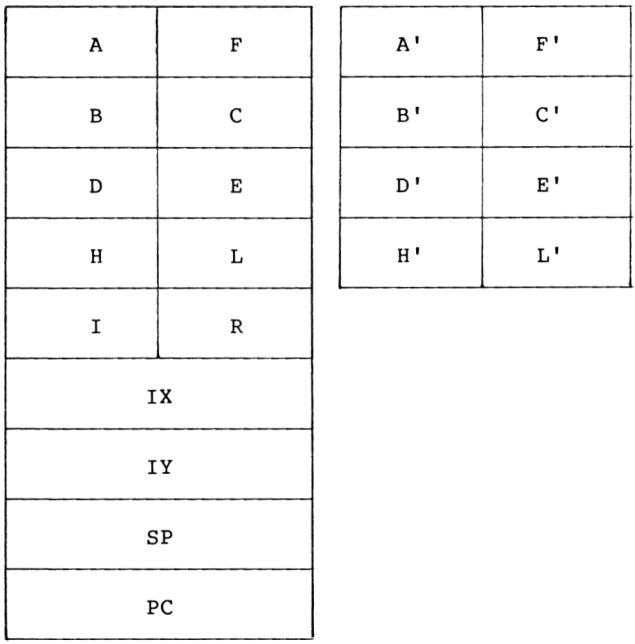
I det følgende skal du indtroduceres overfor de faciliteter der ligger i mikroprocessoren til hjælp for din programmering. Efter disse afsnit starter gennemgangen af instruktionerne.

REGISTRE:

AMSTRADs indre er delt op i flere enheder, hvoraf CPU-en (mikroprocessoren) kun er een af disse. Denne indeholder et slags lager svarende til det lager AMSTRAD i øvrigt har, og som hedder RAM. Mikroprocessorens lager er imidlertid uhyre meget mindre end AMSTRADs ialt 64 K. Mikroprocessorens lager består faktisk kun af 24 celler, hver på 8 binære tal hver. Binære tal skal vi fremtidig kalde bit forkortet fra det engelske BInary digiT. Disse 24 celler er delt op således at nogle hører sammen to og to, andre igen kan både være en selvstændig celle men også del at et par. Endelig er der celler som altid er alene. Hver af sådanne enheder af celler kaldes for registre, og hver register har et bestemt navn. Dette skal vi straks komme tilbage til.

Når vi skal udføre et program udnyttes disse registre til at lagre de tal (maskin KODER) der står for funktionerne der skal udføres. I andre tilfælde peger registre blot på den kode i lageret der skal udføres. Der findes således et ganske specielt register der konstant indeholder adressen på det tal, hvis kode skal udføres. Dette svarer til at en BASIC-variabel konstant indeholdt linienummeret på den BASIC-sætning, der skulle udføres.

Nedenfor er vist de 24 registre, sammensat i de par de tilhører:



Figur .Oversigt over samtlige registre.

I det efterfølgende afsnit bliver registrene gennemgået eet for et. Det er imidlertid vigtig at gøre sig klart at det er disse registre vi skal bruge hver gang vi skal have noget udført. Skal du eksempelvis addere to tal med hinanden så sker dette med brug af disse registre. Det ene tal lagres i et register og det andet i et andet register. Idet der findes en instruktion der kan lægge indholdet i to registre sammen, skal vi blot have den instruktion udført. Skal vi udskrive noget på Monitoren skal vi bruge et register(par) til at pege på adressen hvor vores tekst står lagret, samtidig med at et andet register konstant skal lagres med koderne for de bogstaver og tal vi vil udskrive. Alt dette vil du komme til at se i detalje i det følgende.

En generel ting er dog at vi konstant får behov for at lagre tal i registrene og i AMSTRADs lager. Dette skulle du gerne have prøvet før - og således fundet ud af at det faktisk ikke kan lade sig gøre at lagre et tal større end 255 i een adresse i AMSTRADs

lager. Dette hænger sammen med at hver lagercelle kun indeholder 8 binære tal. Med disse 8 bit er der en grænse for hvor stort et tal man kan lagre. Skal man skrive et større tal må man bruge flere adresser, d.v.s. flere bits.

" A " - REGISTRET:

Dette er hovedregistret. A'et står for "akkumulatoren", d.v.s. den der summerer. Dette er imidlertid langt fra " A "-registrets eneste funktion, men navnet er alligevel godt idet samtlige resultater af de instruktioner der findes lagres næsten alle i dette register. Der er imidlertid undtagelser herfra, men disse vil vi komme tilbage til.

Med " A "-registret kan du f.eks. lægge to tal sammen ved at lagre det ene i registret og så enten

- lagre det andet i et andet register og så fratrække indholdet i de to registre.

- angive tallet der skal fratrækkes " A " direkte.

- angive den adresse hvori tallet står, som skal fratrækkes " A "-registret.

Efter hver instruktion, hvor resultatet lagres i " A "-reigstret er det klart at hvad enten der stod i reigstret forud vil nu blive slettet. Faktisk kunne man sige at en addition kunne foregå som vist nedenfor (i BASIC skrivemåde):

- LET A = A + PEEK (adresse)
- LET A = A + tal
- LET A = A + B

svarende til de tre muligheder beskrevet ovenfor.

" A "-registret kan imidlertid kun indeholde tal op til en vis størrelse - nemlig op til 255. Dette skyldes at registret kun indeholder 8-bit. Det maksimale tal 8-bit kan angiver er 255 svarende til at samtlige bit er "set", d.v.s. 1. Som nævnt tidligere svarer 8-bit til een byte, hvorfor du vil se navne som enkeltbyte- og 8.bit register brugt i flæng.

Arsagen til at indlede med " A "-registret er som nævnt at dette register er udstyret med så langt de fleste muligheder. Der findes således instruktioner som kun kan udføres med " A "-registret som ene part.

En af disse fordele er bl.a. den tætte forbindelse, der er mellem " A " og et andet register - " F "-registret, som også kaldes flag- eller statusregistret. Navnet skyldes den MEGET vigtige rolle dette register spiller i angivelsen af resultater.

Når vi udfører en multiplikation eksempelvis i BASIC og vi f.eks. ønsker at vide om resultatet er nul, så må vi teste indholdet af den variabel, som indeholder resultatet. Se nedenfor:

```
LET P = A * B
IF P = 0 THEN .....
```

I maskinkode kan man udføre nøjagtig det samme, men hvor resultatet blot ikke kan stå i en variabel, idet disse ikke findes, men i stedet skrives i et register eller i en lageradresse. Ønsker vi at vi om resultatet af multiplikationen er blevet nul behøver vi ikke teste på selve registret (eller adressen), men har mulighed for at aflæse en bestemt bit i " F "-registret, hvis eneste opgave er at signalere, når et resultat af en instruktion er blevet nul. Signalet sker ved at bitten går fra et være "reset", d.v.s. 0, til at blive "set", d.v.s. 1.

Hermed er " F "-registret illustreret, idet dette register faktisk indeholder 8 (6) flag til at markere resultaters størrelse m.m. Se nærmere i afsnittet om " F "-registret.

Sammenhørigheden mellem " A " og " F "-registret skyldes således at (næsten) samtlige resultater placeres i " A "-registret, og det er på disse resultater " F "-registret signalerer.

I et tidligere eksempel så vi hvordan " ADD 7 "-instruktionen kunne summere 7 til indholdet til " A "-registret. I denne forbindelse er der klart at med mindre vi kender indholdet i " A " inden additionen - ja, så har vi ikke mulighed for at kende summen. Imidlertid kan vi TESTE på summen, hvilket vi ville gøre med:

```
IF A = tal THEN ...
```

fra BASIC. Da denne sætning ikke findes i maskinkode kunne vi jo bruge " F "-registret. Forestiller vi os at vi gerne vil vide om summen er overskrevet 255 kan dette undersøges via " F "-registret. Dette hænger sammen med at blandt de 8 bit i registret findes een som bliver " set", når resultatet af en instruktion (som ADD) bliver større end et register kan indeholde. Dette er jo netop tilfældet med " A "-registret, der maksimalt kan indeholde tal op til 255. Det drejer sig om første bit, d.v.s. bit nr. 0, i " F "-registret, hvilket også kaldes CARRY-flaget, eller på dansk 'mente-flaget'. Når dette er "set" er der opstået en mente - "reset" er der ingen mente. Vi kan nu ændre instruktioner så vores program tester resultatet i " A " og kun lagrer resultatet i adresse 40000, hvis summen er mindre end eller lig 255:

Assembler	Decimal kode	Kommentar
ADD 7	198 7	læg tallet 7 til indholdet i " A "-registret.
RET C	216	hop tilbage til BASIC, hvis CARRY-flaget er "set".
LD (40000).A	50 64 156	gem indholdet i " A "-registret i adresse 40000.
RET	201	hop tilbage til BASIC.

Figur . Sammenhængen mellem " A " og " F "-registret.

Du skal ikke undre dig formeget over de viste instruktioner, som vi endnu ikke har omtalt. Læs blot kommentarerne og gå videre i teksten.

I figuren nedenfor er vist BASIC-programmet, der POKE'r maskinkoden ind i lageret.

```
Linienr.  Kommando

    10    REM ****** addition *******
    20    FOR z=1 TO 7
    30    READ f
    40    POKE z+41999,f
    50    NEXT z
    60    DATA 198,7,216,50,64,156,201
    70    POKE 40000,0
    80    CALL 42000
    90    IF PEEK(40000)><0 THEN PRINT
          PEEK(40000)
    95    REM ************************
```

Hvis du ikke ser resultatet på skærmen er det fordi summen er overstegen 255.

Læg mærke til hvordan AL maskinkode først skal lagres i adresser i form af talkoder, for derefter at blive udført. Man kan altså ikke bare indtaste instruktionsnavnene, som man kan det i BASIC, og så få instruktionerne udført. Ligeledes kan du heller ikke foranstille instruktionsnavnene et linienummer og så opbygge et BASIC-lignende program (på skærmen). Indledningsvis er det altså en smule mere besværligt både at indtaste og få udført. Det er derfor vigtigt at du stadig bruger BASIC så meget det kan lade sig gøre, og kun de dele af dine programmer der behøver meget høj hastighed, til skærmbehandling, søgninger etc., at disse dele laves i maskinkode. Det vil nemlig ofte tage ca 50 gange så lang tid at lave et program i maskinkode end i BASIC.

Når du har kørt det ovenfor viste program vil du helt sikkert gerne vide, hvordan man får indflydelse på indholdet i " A "-registret, inden additionen. Dette kan du få med en instruktion der hedder LOAD eller forkortet " LD ". Vi har allerede set den i brug, da vi skulle lagre indholdet af " A " i adresse 40000. Læg mærke til at instruktionen således er meget fleksibel, idet man både kan lagre i lageradresser og i registre med " LD "-instruktionen.

Lagring af et tal i " A "-registret kan ske ved instruktionen

```
LD A.tal
```

som læses "skriv 'tal' ind i A". Ligegyldigt hvad der stod i " A "-registret forud bliver det nu slettet og 'tal' skrives oveni. Igen skal du huske at " A " kun indeholder 8 bit, og derfor ikke kan indeholde tal større end 255.

Skal vi nu indføje denne instruktion i vores program kommer det til at således ud:

Assembler	Decimal kode	Kommentar
LD A.100	62 100	skriv tallet 100 ind i " A "-registret.
ADD 7	198 7	læg tallet 7 til indholdet i " A "-registret.
RET C	216	hop tilbage til BASIC, hvis CARRY-flaget er "set".
LD (40000).A	50 64 156	gem indholdet i " A "-registret i adresse 40000.
RET	201	hop tilbage til BASIC.

Figur . Sammenhængen mellem " A " og " F "-registret.

Vi skal så blot HUSKE at ændre BASIC-programmets DATA-sætning tilsvarende:

```
Linienr.  Kommando

   10     REM ****** addition *******
   20     FOR z=1 TO 9
   30     READ f
   40     POKE z+41999,f
   50     NEXT z
   60     DATA 62,100,198,7,216,50,64,156,201
   70     POKE 40000,0
   80     CALL 42000
   90     IF PEEK(40000)><0 THEN PRINT
          PEEK(40000)
   95     REM ************************
```

Figur . BASIC-loader.

Det vil nu være på sin plads at ændre linie 90 så testen ikke bliver så triviel som før. Endvidere har du måske undret dig over hvorfra de 'decimal koder' kommer fra. Bagest i bogen er

vist en tabel, sorteret dels på decimale koder, dels alfabetisk på instruktionsnavn. Heri kan du finde samtlige koder for instruktionerne. Vi skal se på disse og hvordan man angiver adresser etc. i afsnittene, der gennemgår instruktionerne i detalje.

" B " OG " C " REGISTRENE:

Begge registre er hver især enkeltbyte registre, men de kan sættes sammen og derved udgøre et dobbeltbyte register - også kaldet et registerpar. Forbogstaverne B og C kommer af det engelske Byte Counter eller Byte tæller. De hedder således fordi disse to registre enten B alene eller BC sammen er tilknyttet flere instruktioner, hvor de tæller antallet af gange bytes bliver flyttet, sammenlignet, kopieret eller slettet. Disse funktioner er samlet i de der kaldes 'de automatiske instruktioner', som vi alle skal vende tilbage til. (LDIR,DJNZ,CPIR etc.).

Tællere er således disse registres vigtigste funktion. Tællere kender de formentlig allerede fra BASIC i forbindelse med FOR/NEXT sætninger eller løkker (loops) af anden type:

```
      LET a=2
100   LET a=a+1
      IF a>2 THEN GOSUB....
      ...
      IF a>100 THEN GOSUB...
      ...

      ...
      GOTO 100
```

Her optræder variablen a som tæller, og det faktisk i denne rolle du oftest vil se " BC " registerparret.

Når disse instruktioner ikke bruges kan " BC " registerparret bruges til havd du måtte ønske, d.v.s. til midlertidigt lager for mellemresultater eller lig.

" D " OG " E " REGISTRENE:

Disse registre hører sammen svarende til " BC "-registerparret, men kan også bruges hvert for sig. De har fået navnene D og E grundet det engelske Destination eller 'modtagelsesstedet'. Dette skyldes at disse to registre sammen kan indeholde en adresse. Adresser kræver ALTID 16-bits. Denne adresse bruges i mange instruktioner som modtageradresse for tal der skal flyttes fra et til et andet sted. Dette sker i kopieringer, sletninger, flytninger og sammenligninger.

Når disse instruktioner ikke bruges kan " DE " registerparret bruges til havd du måtte ønske, d.v.s. til midlertidigt lager for mellemresultater eller lig.

" H " OG " L " REGISTRENE:

Dette registerpar er det mest fleksible af dobbeltbyte registrene. Der er således knyttet flere instruktioner og muligheder til dette registerpar end til noget andet par. " HL "-registerparret bruges også i mange instruktioner med en bestemt funktion, nemlig som indeholder af 'afsendelsesadressen'. Registerparret bliver herved pegepind til de(n) adresser, der skal flyttes, slettes, kopieres eller sammenlignes. " DE " er modparten, idet dette par indeholder modtageradressen.

Skal du eksempelvis kopiere 100 bytes stående i adresse 20000-20099 til adresserne 30000-30099 findes der en instruktion, der netop er lavet til dette formål (LDIR). Inden denne instruktion skal udføres sørger man for at skrive afsenderadressen i " HL " og modtageradressen i " DE ". Antallet af bytes der skal flyttes lagres i " BC "-registerparret. Udføres instruktionen så vil flytningen ske automatisk.

Det der egentlig sker inde i instruktionen er at først flyttes indholdet i adressen som " HL " indeholder over i adressen som " DE " indeholder. Hvis vi nu har lagret

HL = 20000
DE = 30000

BC = 100

vil adresse 30000 modtage indholdet i adresse 20000. Herefter øges både " DE " og " HL " med een, hvorved de nu peger på henholdvis 30001 og 20001. Processen kan gå om. For at få flyttet netop 100 bytes (tal) er det vigtigt at have en tæller. Dette er jo netop " BC " opgave. Dette betyder at " BC " skal nedtælles med een for hver gang at " DE " (og " HL ") tælles op. Når " BC " er nået nul er vi færdige.

Når disse instruktioner ikke bruges kan " HL " registerparret bruges til havd du måtte ønske, d.v.s. til midlertidigt lager for mellemresultater eller lign.

" IX " OG " IY " REGISTERPARRENE:

Disse registre er begge FØDT 16-bit eller dobbeltbyte registre. De kan hver for sig indeholde 16-bit tal og har egentlig ikke andet til fælles. Du kan således IKKE splitte dem op i to " I ", et " X " og et " Y "-register. Dette ville iøvrigt også skabe forvirring, idet der allerede findes et " I "-register, som vi straks skal se det.

Disse to registerpar kaldes for indeksregistrene på grund af en særlig måde hvorpå man kan lagre tal ind i dem. Princippet kaldes naturligvis for indeksering og sker ved at du til en basisadresse kan summere et tal, hvorved du udpeger en ny adresse. Ved at ændre basisadresse, f.eks. konstant øge den med et tal, kan du adressere adresser i mønstre, f.eks. hveranden. Basisadressen skrives i " IX " eller " IY " og der findes så instruktioner, der tillader FØRST at der adderes et tal til denne basisadresse, DEREFTER at indeholdet i den nye adresse hentes.

" F " - REGISTRET:

Dette register kaldes flag- eller statusregistret. Hvad har flag med sagen at gøre ?. -Jo, eksempelvis bruger skibe hyppigt flag for at meddele andre skibe en eller anden besked. Det gælder f.eks. nationalitet, lasttype, nødhjælp m.m. Meddelelsen vises ved at det pågældende flag hejses til tops på skibet flagmast. På nogenlunde samme vis meddeler AMSTRADs mikroprocessor sig til os via " F "-registret. I dette register findes 8 bit (8 flag) som enten kan være 1 (hejst op) eller 0 (nede). Hermed kan maskinen signalere forskellige ting afhængig af hvilken betydning de forskellige flag har.

De 8 flag bruges i princippet uafhængigt af hinanden til at give information om resultater af de forskellige instruktioner, som vi sætter AMSTRADs mikroprocessor til at udføre. Det er imidlertid ikke alle instruktioner som påvirker flagene, hvilket du vil opdage kan være til fordel i mange tilfælde. Den største gruppe instruktioner der ikke påvirker flagene er de såkaldt lagringsinstruktioner, på engelsk: LOADNING. Vi har allerede se to variationer af denne.

Det er netop af denne grund at " F "-registret ikke bruges som de øvrige registre. Der findes kun ganske få instruktioner som inkluderer hele " F "-registrets 8 bit. Til gengæld er der et hav som tester på et af " F "-registrets flag. Forestil dig f.eks. at vi lagrede et tal i " F "-registret lige efter en addition, hvor vi gerne ville aflæse et af flagene. Dette ville jo gå tabt og vi måtte bygge vores program om - derfor " F "-registret BØR IKKE BRUGES SOM MIDLERTIDIGT LAGER FOR MELLEMRESULTATER ETC.

Det er nu på tide at se på de flag der findes i registret. Selvom der er 8 bit er der dog kun 6 af disse der bruges til flag. Endvidere vil vi kun bruge de 4 (5) af disse flag i denne bog. Det sidste flag benyttes af maskinen internt oftest i forbindelse med en bestemt talskrivemåde kaldet BCD. Flaget der hentydes til er HALF-CARRY:

bit nr.	navn		forkortelse
7	SIGN	(fortegn)	M el. P
6	ZERO	(nulflag)	Z
5			
4	HALF-CARRY	(delmente)	
3			
2	PARITET/OVERFLOW	(paritet/o.)	PE/PO
1	SUBTRAKTION	(fratrækning)	N
0	CARRY	(mente)	C

Figur :Oversigt over " F "-registrets flag.

Som nævnt er der to bits der ikke bruges til flag, nemlig bit nr. 3 og 5. Endvidere vil HALF-CARRY flaget ikke blive omtalt i detalje her i bogen.

Vi skal nu gennemgå flagene et for et og vi starter med SIGN-flaget i bit nr. 7.

SIGN-FLAGET: bit nr. 7.

Sign betyder i denne forbindelse 'fortegn', d.v.s. horvidt et tal er positivt (og nul) eller negativt.

SIGNFLAGET BLIVER "SET", NÅR RESULTATET AF EN ARITMETISK ELLER LOGISK OPERATION BLIVER NEGATIVT. bLIVER RESULTATET NUL ELLER POSITIVT BLIVER SIGNFLAGET "RESET".

Vi skal senere komme tilbage til, hvordan man skriver negative tal i binær kode.

En vigtig ting er at det IKKE er alle instruktioner, der påvirker SIGN-flaget trods det at resultatet bliver negativt el.a. I disse tilfælde vil flaget blot være have det hele tiden har været. Som tidligere nævnt påvirker lagringsinstruktionerne ikke flagene. Dette betyder at selvom du lagrer et negativt til i " A "-registret, så vil SIGN-flaget ikke blive påvirket.

De aritmetiske og logiske instruktioner vil blive gennemgået i et senere afsnit.

SIGN-flaget kan altså anvendes til at finde ud af om resultatet af vores addition fra forrige afsnit er positivt (og nul) eller negativt. Vi kan ved at ændre lidt på dette program bruges SIGN-flaget. I stedet for at retrunere til BASIC, når resultatet overstiger 255, kan vi returnerer hvis resultatet bliver negativt, d.v.s. hvis SIGN-flaget bliver "set". Dette er vist nedenfor:

Assembler	Decimal kode	Kommentar
LD A.tal	62 tal	skriv 'tal' ind i " A "-registret.
ADD 7	198 7	læg tallet 7 til indholdet i " A "-registret.
RET M	248	hop tilbage til BASIC, hvis SIGN-flaget er "set".
LD (40000).A	50 64 156	gem indholdet i " A "-registret i adresse 40000.
RET	201	hop tilbage til BASIC.

Figur . Sammenhængen mellem " A " og " F "-registret.

Vi skal så blot HUSKE at ændre BASIC-programmets DATA-sætning tilsvarende:

```
Linienr.  Kommando

    10    REM ****** addition *******
    20    FOR z=1 TO 9
    30    READ f
    40    POKE z+41999,f
    50    NEXT z
    60    DATA 62,tal,198,7,248,50,64,156,201
    70    POKE 40000,0
    80    CALL 42000
    90    IF PEEK(40000)><0 THEN PRINT
          PEEK(40000)
    95    REM ************************
```

Figur . BASIC-loader.

Alt hvad du skal gøre er at indsætte et tal (på 'tals' plads). Som du kan se lagres summen KUN hvis SIGN-flaget er "reset", d.v.s. hvis summen er positiv.

SIGN-flaget kan, hvad man kalder 'negeres'. På godt dansk kunne man sige: -gøres omvendt-. Normalt bliver flaget "set", når resultatet bliver negativt. Negationen af dette vil være at SIGN-flaget bliver "set", når resultatet IKKE bliver negativt. Dette kan lade sig gøre, som vist nedenfor (på RET xx instruktionen):

Assembler	Decimal kode	Kommentar
LD A.tal	62 tal	skriv 'tal' ind i " A "-registret.
ADD 7	198 7	læg tallet 7 til indholdet i " A "-registret.
RET P	240	hop tilbage til BASIC, hvis SIGN-flaget er "set".
LD (40000).A	50 64 156	gem indholdet i " A "-registret i adresse 40000.
RET	201	hop tilbage til BASIC.

Figur . Illustration af SIGN-flag negation.

Det er måske en lille smule forvirrende, idet programmet nu vil lagre et tal, hvis summen af 7 og " A "-registret bliver negativt !.

ZERO-FLAGET: bit nr. 6.:

ZERO eller 'nul'-flaget er, som navnet antyder, et flag som tester om resultatet i " A "-registret efter en instruktion er blevt nul.

ZEROFLAGET BLIVER "SET", NÅR RESULTATET AF EN ARITMETISK ELLER LOGISK OPERATION BLIVER NUL. BLIVER RESULTATET IKKE NUL GØRES ZEROFLAGET "RESET".

ZERO-flaget fungerer sammen med SIGN-flaget for de instruktioner hvor begge er aktive. der findes nemlig instruktioner der kun påvirker det ene af disse flag. Du må i hvert enkelt tilfælde finde ud af hvilke flag en instruktion benytter hvis du skal bruge flagene.

Igen kan vi ændre vores 'standard' program så vi returnerer, hvis summen bliver nul:

Assembler	Decimal kode	Kommentar

LD A.tal	62 tal	skriv 'tal' ind i " A "-registret.
ADD 7	198 7	læg tallet 7 til indholdet i " A "-registret.
RET Z	200	hop tilbage til BASIC, hvis ZERO-flaget er "set".
LD (40000).A	50 64 156	gem indholdet i " A "-registret i adresse 40000.
RET	201	hop tilbage til BASIC.

Figur . Sammenhængen mellem " A " og ZERO-flaget.

Vi skal så blot HUSKE at ændre BASIC-programmets DATA-sætning tilsvarende:

```
Linienr.  Kommando

    10    REM ****** addition *******
    20    FOR z=1 TO 9
    30    READ f
    40    POKE z+41999,f
    50    NEXT z
    60    DATA 62,tal,198,7,200,50,64,156,201
    70    POKE 40000,0
    80    CALL 42000
    90    IF PEEK(40000)><0 THEN PRINT
          PEEK(40000)
    95    REM ************************
```

Hvis gerne vil have summen til at blive nul skal du tænke på at den maksimale sum " A "-registret kan vise er 255. Bliver summen således 256 stilles samtlige bits i " A " "reset", d.v.s. som 0. Prøv derfor at summere de 7 med 249 !.

ZERO-flaget kan, hvad man kalder 'negeres'. På godt dansk kunne man sige: -gøres omvendt-. Normalt bliver flaget "set", når resultatet bliver nul. Negationen af dette vil være at ZERO-flaget bliver "set", når resultatet IKKE bliver nul. Dette kal lade sig gøre, som vist nedenfor (på RET xx instruktionen):

Assembler	Decimal kode	Kommentar
LD A.tal	62 tal	skriv 'tal' ind i " A "-registret.
ADD 7	198 7	læg tallet 7 til indholdet i " A "-registret.

```
RET  NZ          200          hop tilbage til BASIC, hvis
                              ZERO-flaget er "set".
LD   (40000).A   50 64 156    gem indholdet i " A "-
                              registret i adresse 40000.
RET              201          hop tilbage til BASIC.
```

Figur . Illustration af negation af ZERO-flag.

Det er måske en lille smule forvirrende, idet programmet nu vil lagre et tal, hvis summen af 7 og " A "-registret bliver nul !.

CARRY-FLAGET bit nr. 0:

Carry betyder mente, d.v.s. en angivelse af når der mellem bytes er lånt eller mangler een. Vi skal derfor igen igang med at se på hvor store tal een og to bytes kan indeholde, og havd der sker når man overskrider disse grænser eksempelvis ved addition.

Enkelt bytes, d.v.s. 8-bit tal, kan som nævnt KUN indeholde tal mellem 0 og 255 (regnet uden fortegn). Idet samtlige lageradresser kun indeholder 8-bit kan hver af disse således kun indeholde tal mellem 0 og 255.

Skal to enkelt bytes derfor lægges sammen kan resultatet jo kræve een eller to bytes afhængig af hvor store de to addender er. To bytes hvis sum er større end 255 kræver således to bytes. Når man skal lave et program der skal addere to tal er det således altid summens overgrænse, d.v.s. summens maksimale værdi, der kan volde problemer. Når man har fundet antallet af bytes der skal til at skrive dette resultat vil resten løse sig selv. Når to bytes skal summeres må overgrænsen være hvor begge antager DERES største værdi. Dette må betyder at overgrænsen ved addition af to bytes er:

255 + 255 = 510.

For at skrive tallet 510 i en lageret behøves TO bytes. Dette kan du forsikre dig ved at studere hvad det betyder at have den 9., 10., 11., osv bit lagt til de første 8. Gør du dette finder du ud af at det faktisk kun er den første bit af den anden byte, der er brug for til at skrive tallet 510:

bit nr.	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
værdi																
510								1	1	1	1	1	1	1	1	0

Da et af flagene, nemlig CARRY, registrer når et resultat ikke længere kan stå i de tilrådigt stående bits. Hvis vi tæller dette flag med kan en byte indeholde tal op til 256 og summen af to bytes kan dermed komme op på 512. Dette resultat fåes imidlertid kun ved at 'fuske' med de to addender inden de adderes.

Men lad os se på nogle praktiske eksempler. Lad os sammenligne to addition, den ene hvor resultatet kun kræver een byte, den anden hvor resultatet kræver to. Vi adderer 65 med 64 samt 132 og 135:

ex. 1

decimal	(carry)	binær
64	-	0100 0000
+ 65	-	0100 0001
129	0	1000 0001

ex. 2

decimal	(carry)	binær
132	-	1000 0100
+ 135	-	1000 0111
267	1	0000 1011

Her ser du hvordan additionen påvirker CARRY-flaget. Da resultatet af additionen ALTID ligger i " A "-registret er det naturlig at kontrollere dette register for om summen er korrekt. Hvis vi ser på første eksempel bliver summen 129. Kontrollerer vi nu de "set"'te og "reset"'te bits, som angivet, passer resultatet også i den binære skrivmåde. Ser du derimod på andet eksempel passer de "set"'te og "reset"'te bits imidlertid IKKE med resultatet 267. Dette skyldes at når vi kontrollerer " A "-registret ser vi ikke CARRY-flaget samtidig. Flaget er kun medtaget her for illustration. Egentlig står der således KUN at summen bliver 11 ?. Kan dette nu passe ? - vi tester det ved at opstille et program der kan beregne summen for os:

Assembler	Decimal kode	Kommentar
LD A.132	62 132	skriv 132 ind i " A "-registret.
ADD 135	198 135	læg tallet 135 til indholdet i " A "-registret.
LD (40000).A	50 64 156	gem indholdet i " A "-registret i adresse 40000.
RET	201	hop tilbage til BASIC.

Figur . Test af addition mellem 132 og 135.

BASIC-programmet til lagring og udførsel:

```
Linienr.  Kommando

   10   REM ****** addition *******
   20   FOR z=1 TO 8
   30   READ f
   40   POKE z+41999,f
   50   NEXT z
   60   DATA 62,132,198,135,50,64,156,201
   70   POKE 40000,0
   80   CALL 42000
   90   PRINT PEEK(40000)
   95   REM ************************
```

Figur . BASIC-loader.

Når du har 'kørt' dette program skulle du gerne se at summen tilsyneladende er de 11 ?. Du har formentlig set at vi mangler angivelsen af at CARRY-flaget signal. Dette betød jo netop at der manglede plads i de 8 bit. Når vi indregner dette flag, som jo bliver bit nr. 8 får vi det korrekte resultat. Bit nr 8 har jo værdien 256 (2 opløftet i 8 potens eller 2 gange med sig selv 8 gange - prøv selv efter). Summen af 256 og vores resultat 11 giver netop det rigtige - 267 -.

Vi har indtil nu kun set på enkeltbyte tal. Men naturligvis skal vi også se på større tal. Udvidelsen fra een til dobbelt byte tal kan tilmed overføres til yderligere udvidelser til tal krævende 3, 4 og 5 bytes. Vi stopper dog ved de 2 byte, da vi også vil få brug for dette i anden sammenhæng - nemlig når vi skal arbejde mere detaljeret med adresser.

Ligegyldigt hvormange bytes vi arbejder med pr. tal vil din maskine altid kunne arbejde med dem. Dette til trods for at AMSTRAD er en 8-bit baseret computer, d.v.s. den bearbejder kun 8-bit ad gangen. Netop ved at opdele og behandle een byte ad gangen kan vi arbejde med vilkårligt store tal. Opdelingen kræver imidlertid omtanke samt det at tiden maskinen vil være om at addere større tal vil stige.

Vi skal således se på 16-bit tal, d.v.s. tal af størrelsen mellem 0 og 65536. CARRY-flaget får her samme betydning som ovenfor. Dette kan illustreres med addition af dobbelt byte tal såsom 1000 med 2000. Lad os først se den binære skrivemåde for de to tal:

decimal	binær	
	low byte	high byte
1000	1110 1000	0000 0011
2000	1101 0000	0000 0111

Bemærk skrivemåden. Et dobbeltbyte tal kræver 16-bit. Da AMSTRADs lager kun indeholder 8-bit pr. lagercelle - opdeler man de 16-bit i 2 gange 8. Da bit 0 til 7 og 8 til 15 skal holdes strengt adskildt og ikke sammenblandes kalder man bit 0 til 7 for LOW-byten og 8 til 15 for HIGH-byten. 'High' fordi disse øverste 8 bits indeholder den største del af tallet.

Du kender formentlig problemstillingen fra BASIC, hvor man ønsker at lagre et tal større end 255. Man må derfor splitte tallet op, hvilket normalt sker med formelen:

```
POKE adresse,INT (tal/256)

POKE adresse+1,( tal/256-PEEK(adresse))*256
```

og herved har man allerede opsplittet tallet i et high- og en low del. Den øverste er highdelen.

Det har altså IKKE noget at gøre med at highdelen er et større tal end lowdelen - det kan udemærket være mindre. Der er meningen med tallet, d.v.s. den værdi vi giver det, der tæller. Eksempelvis vil tallet 1000 bestå af lowbyte og highbyte:

```
highbyte :  INT (1000/256) = 3
low-byte :  (1000/256 - 3)*256 = 232
```

Her er den talmæssige værdi af lowbyten 232 og highbyten 3. MEN stadigvæk betragter vi highbyten for størst. Når man har valgt denne opsplitning kommer hver ener i highbyten til at stå for 256'ere. De 3 betyder således 3*256 = 768. Lowbytens enere betyder stadig enere, hvorfor den stadig har værdien 232.

Tilbage til additionen af 1000 og 2000. Skal vi nu sørge for at additionen går godt skal vi bruge en ny instruktion som kan summere 16-bit tal. Denne instruktion hedder " ADC ". Hvis du vil se additionen er denne vist i afsnittet for denne instruktion. Senere i bogen vil du også finde multiplikation, division og subtraktion m.m.

SUBTRAKTIONS FLAGET bit nr. 1

Dette flag bruges alene til at markere at der er udført en subtraktionsinstruktion. Flaget bliver kun påvirket, hvis der udføres en 'reel' subtraktion. Dette vil vi komme tilbage til i forbindelse med sammenligningsinstruktionerne.

SUBTRAKTIONSFLAGET BLIVER "SET", HVIS DEN NETOP UDFØRTE INSTRUKTION VAR EN SUBTRAKTION. VAR DETTE IKKE TILFÆLDET BLIVER FLAGET "RESET".

Ovennævnte regel gælder naturligvis KUN for instruktioner, der påvirker flaget.

PARITET/OVERFLOW-FLAGET bit nr. 2.

Dette flag har egentlig to funktioner. Dels at markerer, når der forekommer lige/ulige paritet, dels når der er binær overløb. De to funktioner skulle ikke kunne forstyrre hinanden, idet de optræder på forskellige instruktioner.

Pariteter vedrører udelukkende de man kalder logiske instruktioner. Se herom. At et binært tal har lige paritet betyder at antallet af "set"te bits er lige. Er et ulige antal bits "set" har tallet ulige paritet. Begrebet benyttes også i forbindelse med datatransmission, hvor man sender data fra et til et andet sted. Pariteten kan da fungere som kontrol for at du har fået sendt det rigtige antal bits afsted.

Overflow eller - overløbs - funktionen fungerer ligsom CARRY blot ved at, når resultatet af en operation kræver mere end de 7 første bits gøres OVERFLOW "set". det er imidlertid begrænset det antal instruktioner der benytter denne facilitet. Den er dog uundværlig når man begynder at arbejde med fortegn for sine binære tal. Fortegnet lagres da som bit nr. 7. Et signal om at bit nr. 6 er overskredet er da en vigtig information, for ikke at misopfatte fortegnet.

HALF-CARRY FLAGET bit nr. 4:

Dette flag svarer til CARRY, men bliver "set", når bit nr 3 overskrides for at kunne angive resultatet af en operation. Denne facilitet benyttes meget i en bestemt skrivemåde kaldet BCD.

AFTRYKS - REGISTRENE:

Disse registre er de mindst fleksible, der findes blandt de ialt 24 enkeltbyte registre. Dette hænger sammen med at kun ganske få instruktioner er tilknyttet og kan arbejde med disse registre. De bruges alene som midlertidigt lagersted for mellemresultater og dette kun i ekstraordinære tilfælde. De kan nemlig KUN udvekle data med de primære registre omtalt foran. Til hvert primært register findes der således et aftryksregister. Du kan så foretage et aftryk af de primære registre i aftryksregistrene for senere at ville have dette aftryk tilbage.

Med ordet primære menes registrene A, F, B, C, D, E, H og endelig L-registrene. Diagrammet nedenfor viser såvel de primære som aftryksregistrene.

STAK - PEGER REGISTRET:

" SP " eller stakpegeren er et 16-bit register, der har en meget speciel funktion i relation til lagring. De 16-bit skal bruges til at lagre en adresse, og denne adresse bliver konstant 'holdt vedlige' af AMSTRAD.

Normalt angiver man ved lagring i AMSTRADs RAM-lager, hvilken adresse man ønsker at lægge sine tal i, som f.eks.

LD (40000).A

men der findes en anden metode at lagre på. Denne benytter et begreb man kalder stakken. Ordet stakken bruges for at angive at de lagrede tal bliver lagret oven på hinanden efterhånden som de bliver lagt ind på stakken. Natruligvis kan man også tage tal fra stakken, når man får behov for det. Det lidt særprægede ved staklagrings-metoder er at man kun kan få fat i det nederst lagrede tal. Nu sker lagringen oppefra og nedefter, forstået således at hvis du skal lagre to tal, så kommer det sidst lagrede tal til at ligge nederst på stakken. Hvad bruges nu stakpegeren til. Jo denne adresse peger konstant på den næste frie plads til næste lagring. Hver gang et tal der lagret på stakken trækkes der een fra adressen i " SP ". Omvendt bliver der lagt een til " SP ", når et tal tages af stakken. Da vi således kun har adressen på den nederste i stakken, er dette baggrunden for at man siger at det kun er muligt at hente den nederste eller det sidst lagrede tal. Det kan imidlertid udemærket lade sig gøre at hente tidligere lagrede tal, men man må så først have alle de foranliggende tal fjernet.

Lidt simpelt kunne man sige at stakken fungerer som et blindt togspor. Hver togvogn der køres ind på sporet kan kun kommer tilbage ad samme spor, når de skal bruges igen. Hermed skulle du kunne se problemstillingen med at skulle have fat i den forreste eller næstforreste togvogn.

Der knytter sig flere instruktioner til at lagre tal på og hente tal fra stakken. Dette gælder f.eks. " PUSH " og " POP " m.fl.

" R " - REGISTRET:

R står for Refresh eller vedligeholdelse. " R "-registret deltager i vedligeholdelsen af AMSTRADs lager, som konstant bliver gennemstrømmet (elektrisk) for ikke at miste indholdet heri. " R "-registret fungerer som en tæller, der opdateres for hver gang, der hentes instruktioner i AMSTRADS lager. Registret kan ikke umiddelbart anvendes i din maskinkodeprogrammering grundet den konstante opdatering. Måske som tilfældighedsgenerator.

" I " - REGISTRET:

I'et står for "interrupt" eller afbrydelse. Normalt indeholder "I"-registret værdien 63, som medvirker som del af en adresse, der anvendes i AMSTRADs interruptsystem.

Da operativsystemet udnytter interruptsystemet ekstensivt kan registret ikke anvendes som de øvrige registre, d.v.s. som midlertidig mellemlager etc.

" PC " - REGISTRET:

For konstant at holde kontrol med adressen på den næste instruktion, der skal udføres, indeholder dette register 16-bit, der konstant styrer fra hvilken adresse næste instruktion skal hentes til udførelse. Registret kan derfor heller ikke bruges i vores programmer til lagring af data. Registret er imidlertid det bankende hjerte i maskinen og kan på mangfoldige måder øge effektiviteten i dine programmer, når du lærer at udnytte dette register.

LAGRING AF KONSTANTER I REGISTRE:

Når vi skal arbejde med tal og bogstaver i vore maskinkodeprogrammer er det nødvendigt at have noget, der hvori vi midlertidigt kan lagre tal, mellemresultater osv. Dette kan vi gøre såvel i AMSTRADs lager som i maskinens mikroprocessor. Som du allerede har set indeholder mikroprocessoren et antal registre, som vi kan bruge til vore programmer. Det drejer sig f.eks. om A, B, C osv.

I dette og de 3 følgende afsnit skal vi se på forskellige måder at lagre tal i registre. De resterende afsnit i dette kapitel ser på måder at lagre i AMSTRADs lager.

Dette første afsnit omhandler den direkte lagring af tal i 8-bit registre. Ordet "direkte" antyder at du nævner det tal, der skal lagres i 8-bit registret, umiddelbart efter navnet på registret. Dette kunne eksempelvis være:

LD A.8

som lagrer tallet 8 i " A "-registret, som er det mest benyttede 8-bit register. Linien skal læses på følgende måde: " skriv tallet 8 i " A "-registret ". Simpelt IKKE.

Forkortelsen - LD - står for det engelske ord LOAD eller på dansk 'at lagre'. Vi kalder forkortelsen for operationskoden eller assemblerkode. Hele linien kaldes en instruktion.

Ofte kaldes den gruppe af instruktioner, der udfører denne form for lagring for " absolut adressering ". Dette er imidlertid blot et andet navn for det samme.

Der findes som nævnt i bogen første afsnit 7 umiddelbart anvendelige 8-bit registre. Disse var A, B, C, D, E, H og L. Samtlige kan lagres med tal som det er vist for " A "-registret ovenfor. Der er imidlertid en begrænsning på hvor stort et tal der kan lagres i et 8-bit register. Dette skyldes jo at et sådant register kun har 8 bit at vise et tal med. Det største tal bliver som vi har set tidligere - 255.

De 7 instruktioner der findes til direkte lagring af tal i 8-bit registre er vist nedenfor i tabellen:

Tabel 1. Direkte lagring i 8-bit registre.

Assembler	Decimal kode
LD A.n	62 n
LD B.n	6 n
LD C.n	14 n
LD D.n	22 n
LD E.n	30 n
LD H.n	38 n
LD L.n	46 n

Du har nu set de 7 første instruktioner og jeg nu bede dig om at læse det følgende omhyggeligt igennem for fra starten at undgå misforståelser.

Bemærk for det første at instruktioner såvel som registre er skrevet med STORT. Dette vil kendetegne disse i hele bogen. Det lille 'n' er således IKKE et register - ej heller en instruktion. Det er vigtigt at undgå denne misforståelse fra starten.

Læg endvidere mærke til at du kan genfinde 'n' i kollonnen for decimale koder. Dette betyder at du, når du skriver programmer med instruktioner med 'n', så skal du INDSÆTTE et tal for dette bogstav. Endvidere er det vigtigt at sige at et ENKELT 'n' betyder at 'n' skal udskiftes med et tal i størrelsen svarende til et 8-bit tal. Således et tal mellem 0 - 255. Er der to 'n'er betyder det 16-bit tal. Herom senere.

Vi kan lære anvendelsen af den direkte lagrings instruktion ved et par eksempler. Nedenfor lagres registrene " B " og " C " med tal. Til dette skal vi bruge instruktionerne vist i tabel 1:

Assembler	Decimal kode
LD B.200	6 200
LD C.10	14 10
RET	201

Programmet kan først udføres, når det er blevet lagret i AMSTRADs lager. Lad imidlertid være med at forsøge at indtaste instruktionerne til maskinen, som du f.eks. gør med BASIC-kommandoer. AMSTRAD vil ikke kunne forstå (de fleste af) instruktionerne. Dette hænger jo sammen med at alt hvad du indtaster til maskinen vil blive FORTOLKET, som BASIC-kommandoer, hvilket det jo ikke er. Nej, måden at få programmet ind i maskinen er at lagre DE DECIMALE værdier i lageret. Dette gøres

med " POKE "-kommandoen. Nedenfor er vist et eksempel.

```
Linienr.  Kommando

    10    REM**** BASIC loader for program *****
    20    POKE 20000,6
    30    POKE 20001,200
    40    POKE 20002,14
    50    POKE 20003,10
    60    POKE 20004,201
    70    REM************************************
```

Figur 1.BASIC-loader til maskinkode program.

Bemærk at hvis vores maskinkodeprogram består af mange instruktioner vil antallet af " POKE "-kommandoer være stort. Det kan derfor være hensigtsmæssigt i større programmer at benytte

READ / POKE / DATA

kombinationen. Denne vil du se flere gange i de kommende programmer.

Når maskinkode programmet skal lagres indtaster du, som sædvanlig RUN til start af BASIC-programmer. Dette starter imidlertid IKKE maskinkode programmet. Dette sker med

CALL adresse

kommandoen. 'adresse' udskiftes med den adresse vi ønsker at starte maskinkodeprogrammet fra. I vores eksempel vil det være 20000, d.v.s. CALL 20000.

Denne kommando vil starte vores maskinkode program. Først lagres " B "-registret med 200, dernæst " C " med 10. Den sidste instruktion svarer til BASIC-kommandoen

RETURN

hvilket betyder at AMSTRAD 'vender tilbage' til BASIC-niveau. Dette kaldes således, idet det er her at dine BASIC-kommandoer vil blive udført. Hvis vi ikke brugte " RET "-instruktionen ville AMSTRAD aldrig komme tilbage så vi igen kunne indtaste noget. Muligvis vil maskinen 'crashe', så vi må slukke og tænde for den igen for at få den til at udføre ordre.

Selvom programmet foretager det ovenfor nævnte vil du ikke se noget tegn på dette på skærmen. Dette skyldes at der ikke udskrives noget på denne. Senere skal vi imidlertid se programmer anvende såvel skærm som højtaler.

Der er i øvrigt ingen restriktioner for, hvor i AMSTRADs RAM-lager du kan lægge dine maskinkodeprogrammer. Du skal dog være opmærksom på at et eventuelt indtastet BASIC-program kan slette dit maskinkodeprogram, hvis dette ligger i 'BASIC-området'. Nu findes der en BASIC-kommando, som kan mindske BASIC-området. Denne kommando er

MEMORY adresse

som fastsætter den største adresse, op til hvilken BASIC-programmer kan 'vokse'. Sletningen af et maskinkodeprogram, der ligger i BASIC-området sker ved at du fortsætter med at tilføje linier til dit BASIC-program. Herved bruger BASIC-programmet mere og mere af lageret og kan til sidst overskrive dit maskinkodeprogram.

Overskrivningen kan være katastrofal, hvis du efter indtastningen af et større BASIC-program, skal have et maskinkode program udført. Da programmet kan være helt eller delvis overskrevet er det ikke godt at vide hvordan programmet vil 'opføre' sig. Igen er chancen for at AMSTRAD crasher STOR. Igen må du så slukke og tænde for maskinen for at kunne bruge den igen. Hermed mister du dit BASIC-program (og maskinkodeprogrammet). Det kan derfor ikke siges ofte nok - HUSK altid at tage en SIKKERHEDSKOPI af dine programmer INDEN du får dem udført. Dette er jo lettet en smule med den indbyggede cassettebåndoptager der i forvejen sidder i AMSTRAD.

Med MEMORY kommandoen kan du 'sænke' det loft der ligger på BASIC-området. Hermed får du frigivet lagerplads, som du kan bruge til dine maskinkodeprogrammer uden at disse bliver overskrevet af BASIC-linier.

Når du får udført dit BASIC-program er der mulighed for at benytte <BREAK> tasten som nødbremse. Løber dit program ind i en uendelig løkke vil <BREAK> kunne stoppe udførelsen af programmet. Dette er imidlertid ikke tilfældet for maskinkodeprogrammer, der udføres. Disse har du ikke umiddelbart mulighed for at standse med <BREAK>. Du har dog altid muligheden for at forberede din programmering for afbrydelser.

For fuldstændighedens skyld skal det siges at AMSTRAD udmærker sig ved netop at kunne udføre, hvad man kalder

TRAP-PROCESSING

hvilket omfatter mulighederne for INDEN for et BASIC-program at redde sig ud af (visse) opståede fejl, eksempelvis syntaxfejl, indtastningsfejl etc. Dette sker som nævnt i maskinens manual med

ON ERROR GOTO

ON BREAK GOSUB etc.

som giver os mulighed for at undgå at få programmet standset. Den detaljerede anvendelse af disse kommandoer hører imidlertid hjemme i en BASIC-brugervejledning, hvorfor vi ikke går videre med emnet her.

LAGRING AF KONSTANTER I REGISTERPAR:

Forrige afsnit gav kun mulighed for at lagre 8-bit tal. I dette afsnit skal vi imidlertid se på instruktioner, der kan lagre 16-bit tal - IKKE i 8-bit registre - MEN i 16-bit- eller registerpar.

Det er dog muligt at bruge instruktionerne fra forrige afsnit til afløsning af instruktionerne i denne gruppe. Dette skal vi vende tilbage til senere i afsnittet.

Lagringen af 16-bit tal kan illustreres ved at lagre tallet 1000 i " BC "-registerparret. Dette sker med instruktionen

LD BC.n n

hvor det første 'n' betegner lowbyten og det andet highbyten.

Beregningen af low- og highbyten sker med formlerne:

HIGH BYTE: INT (tal / 256)

LOW BYTE: ((tal / 256) - HIGH BYTE) * 256

eller i vort tilfælde, hvor 'tal' skal erstattes af 1000:

high byte: INT (1000 / 256) = 3

low byte: ((1000 / 256) - 3) * 256 = 232

Idet low byten skulle stå først kommer instruktionen til at se således ud:

LD BC.1000 = LD BC. 232 3 = 1 232 3

da operationskoden for " LD BC.n n " er " 1 n n ".

Når du lagrer tal i et registerpar svarer det egentlig til at du foretager to enkelt register lagringer. Det ovennævnte eksempel svarer nøje til nedenstående instruktioner, som kun benytter instruktioner fra den forrige gruppe:

Assembler	Decimal kode
LD B.3	6 3
LD C.232	14 232

men læg mærke til at denne alternative måde at lagre på kræver 4 bytes, d.v.s. 4 decimale tal. Registerpar lagringen krævede een mindre. Jo færre decimale koder en instruktion er opbygget af des kortere tid er den om at blive udført (normalt). Udførelsestiden måles i hvad der kaldes 'clock pulser' eller 'cycles'. Dette er en måle enhed ligesom sekunder er det. Antallet af cycles er altså et mål for hvor længe en instruktion er om at blive udført. Jo lavere tal des hurtigere udføres instruktionen.

En direkte lagring i et 8-bit reister tager 7 cycles, mens en direkte lagring i et 16-bit registerpar tager 10. Men da vi skal udføre den direkte lagring i såvel " B ", som " C "-registrene bliver lagringen i registerparret hurtigst.

Udførelseshastigheden er for alvor vigtig i større maskinkodeprogrammer, men normalt vil det at et program er skrevet i maskinkode kunne øge hastigheden mange gange.

Der findes 6 instruktioner instruktioner til lagring i de 6 registrepar som AMSTRADs mikroprocessor har:

Tabel 2. Direkte lagring i registerpar.

Assembler	Decimal kode
LD BC.n n	1 n n
LD DE.n n	17 n n
LD HL.n n	33 n n
LD IX.n n	221 33 n n
LD IY.n n	253 33 n n
LD SP.n n	49 n n

Her er det vigtigt at bemærke at det tal, der skal lagres, skal opdeles i en high og en low byte. Dette blev vist indledningsvis. Når dette er gjort SKAL

FØRSTE 'n' VÆRE LOW BYTEN

ANDEN 'n' VÆRE HIGH BYTEN

Det skal altså gøres modsat det vi gør i vores talsystem, hvor vi skriver 10'ere længere mod venstre end 1'ere. Du vil imidlertid få STORE problemer, hvis du glemmer denne omvendte sammenhæng, når du arbejder med maskinkodeprogrammering.

Når du ser på instruktionerne til denne gruppe svarer de til den forrige gruppe, hvad angår operationskodens navn m.m. Bemærk imidlertid at det kun er 4 ud af de 6 nye instruktioner der er 'besparende' på antallet af bytes i forbindelse med lagring af tal. De to instruktioner, der findes for indeksregistrene er ikke besparende.

Afslutningsvis skal det nævnes at det kun er de registerpar du ser i nedenstående tabel, som kan bruges i forbindelse med instruktionerne i denne gruppe. Der findes således ingen instruktioner, som kan lagre i " AF "-registre'parret'. Det kan heller ikke lade sig gøre at lagre tal med instruktionerne i to enkeltbyteregistre, som ikke hører sammen i et par. Eksempelvis kan

```
LD CH.n n
```

ikke lade sig gøre. I stedet kan man selvfølgelig bruge instruktionerne, der lagrer i 8-bit registre:

```
LD C.n
LD H.n
```

hvad man så vil bruge dette til.

INSTRUKTIONER TIL INDIREKTE REGISTERLAGRING

Forrige afsnits instruktioner gav os mulighed for at skrive tal i (næsten) alle registre -såvel 8- som 16-bit. Skulle vi f.eks. som vi så det i eksemplet skrive tal i såvel " B ", som i " C "-registret kunne vi gøre dette med en direkte adresseringsinstruktion, hvilket dog ville kræve 2 bytes for hver register.

I dette afsnit skal vi se på instruktioner som gør det muligt at lagre et register med indholdet fra et andet. Lagrer du f.eks. et tal i eet register, således f.eks.

```
LD H.100
```

og skal dette tal også til et andet register kunne du

1) bruge en instruktion, der lagrer registret direkte med tallet, f.eks.

```
LD B.100
```

eller bruge en instruktion fra denne nye gruppe, hvorved

2) du blot nævner de to registre samt LD:

```
LD B.H
```

Dette kan ske såvel for 8-bit registrene som 16-bit registerparrene.

Arsagen til at instruktionerne i denne gruppe kaldes "indirekte" hænger sammen med at du ikke nævner det tal du kopierer fra det ene til det andet register.

1. LAGRING MELLEM 8-BIT REGISTRE:

Den generelle udformning af instruktionen er

LD 8-bit register.8-bit register

hvor 8-bit register står for A, B, C, D, E, H og L. Registret der er nævnt sidst i instruktionen kopiere sit indhold over i det først nævnte register. Decimal koderne for instruktionerne ses nedenfor, opstillet efter 'modtager' registret, d.v.s. det register der skal modtage en kopi.

Tabel . Indirekte lagring i " A "-registret.

Assembler	Decimal kode
LD A.B	120
LD A.C	121
LD A.D	122
LD A.E	123
LD A.H	124
LD A.L	125
LD A.A	127

Tabel . Indirekte lagring i " B "-registret.

Assembler	Decimal kode
LD B.B	64
LD B.C	65
LD B.D	66
LD B.E	67
LD B.H	68
LD B.L	69
LD B.A	71

Tabel . Indirekte lagring i " C "-registret.

Assembler	Decimal kode
LD C.B	72
LD C.C	73
LD C.D	74
LD C.E	75
LD C.H	76
LD C.L	77
LD C.A	79

Tabel . Indirekte lagring i " D "-registret.

Assembler	Decimal kode
LD D.B	80
LD D.C	81
LD D.D	82
LD D.E	83
LD D.H	84
LD D.L	85
LD D.A	87

Tabel . Indirekte lagring i " E "-registret.

Assembler	Decimal kode
LD E.B	88
LD E.C	89
LD E.D	90
LD E.E	91
LD E.H	92
LD E.L	93
LD E.A	95

Tabel . Indirekte lagring i " H "-registret.

Assembler	Decimal kode
LD H.B	96
LD H.C	97
LD H.D	98
LD H.E	99
LD H.H	100
LD H.L	101
LD H.A	103

Tabel . Indirekte lagring i " L "-registret.

Assembler	Decimal kode
LD L.B	104
LD L.C	105
LD L.D	106
LD L.E	107
LD L.H	108
LD L.L	109
LD L.A	111

Instruktionerne har mange anvendelser specielt kan du bruge dem til at lagre et tal midlertidigt, hvis et bestemt register, der indeholder tallet, skal bruges i anden sammenhæng. Dette er vist nedenfor:

Assembler	Decimal kode
LD A.100	38 100
...	
LD L.A	111
ADD 100	198 100
...	
...	
LD A.L	125

Figur .Illustration af en 'RESTORE'-operation.

Ovenfor skal indholdet i " A "-registret bruges to gange, hvilket er en situation du vil komme ud i flere gange. Det viser sig ofte at man ikke blot kan lade indholdet i " A " gå til grunde i første anvendelse, og så lagre værdien ind igen. Dette kan i mange tilfælde IKKE ske.

Dette problem er løst ved at benytte " L "-registret som midlertidigt lager.

FLAG-EFFEKT:

Ingen af de gennemgåede instruktioner påvirker " F "-registrets flag.

Ud over de viste instruktioner findes der 4 som egentligt hører til gruppen, men som bruges temmelig sjældent. Dette skyldes, at de registre der benyttes (det ene af dem) benyttes i forbindelse med styring af INTERRUPT adresser samt vedligeholdelse af AMSTRADs lager. Det drejer sig om " I " og " R "-registrene.

Tabel . Specialiserede instruktioner til indirekte lagring mellem " A ", " I " og " R "-registrene.

Assembler	Decimal kode
LD A.I	237 87
LD A.R	237 95
LD I.A	237 71
LD R.A	237 79

Som nævnt i gennemgangen af mikroprocessorens registre er " R "-registret del af hardware styringen i forbindelse med udførelsen af hver instruktion. Dette skyldtes at registrets første 7 bit udgør en tæller, der konstant øges med een for hver instruktion, der udføres. I hardware terminologi opdeler man udførelsen af en instruktion i flere aktiviteter:

1) FETCH
2) DECODE
3) EXECUTE

hvor den første aktivitet netop er den der øger " R "-registret med een. Aktiviteterne er henholdsvis, hentning af den binære kode som instruktionen består af; omformning af koden til en udførbar instruktion og endelig selve udførelsen af instruktionen.

" R "-registrets 7 første er som nævnt en tæller af antal udførte FETCH-aktiviteter. Den sidste bit anvendes ikke og vil forblive enten 'set' eller 'reset' afhængig af hvad den oprindelig blev lagret med. Vi kan teste dette med nedenstående program, som aflæser " R "-registret et antal gange og for hver gang lagrer indholdet i registret i AMSTRADs lager:

```
                Assembler       Decimal kode

                LD BC.tæller    1 200 0
                LD HL.start     33 32 78
loop:           LD A.R          237 95
                LD (HL).A       119
                INC HL          35
                DJNZ loop       16 250
```

Figur .Program til illustration af " R "-registrets indhold.

Da vi endnu ikke har gennemgået alle de viste instruktion vil vi ikke gennemgå disse i detalje. Du kan eventuelt vende tilbage til programmet, når du har lært alle instruktioner.

Programmet lagrer indholdet i " R "-registret over i " A " som derved kan lagres 'ned' i AMSTRADs lager. 'tæller' hentyder til at vi holder styr på antallet af gange " R "s indhold lagres - i programmet 200 gange. 'start' hentyder til den lageradresse hvorfra vi starter at lagre " R "s indhold.

BASIC-loaderen er vist nedenfor, men indeholder også udførelsen samt PEEK-kommandoerne til at undersøge lagerceller vi lagrer med indholdet fra " R "-registret.

```
Linienr. Kommando

   10   REM *** R-registrets indhold **
   20   FOR a=1 TO 13
   30   READ f
   40   POKE a+42999,f
   50   NEXT a
   60   DATA 1,200,0,33,32,78,237,95,119,
             35,16,250,201
   70   CALL 43000
   80   FOR a=20000 TO 20200
   90   PRINT a,PEEK (a)
   95   NEXT a
   96   REM ****************************
```

Figur . BASIC-loader til ovenfor viste program.

Når du udfører programmet med RUN, skulle du gerne se adresserne 20000 og fremefter 'rulle' ned over skærmen efterfulgt af indholdet i de viste adresser. Bemærk lighederne mellem tallene i adresserne. Hveranden ender på samme tal. Når tallet overstiger 127 begyndes forfra. Disse tal er alle indhold i " R "-registret, men fra forskellige tidspunkter, nemlig tidspunkter adskildt med den tid det tager at udføre instruktionerne i programmet vi har lavet. Antallet af decimale koder i vores programs loop er 6, hvilket skulle svare til forskellen mellem de tal du ser på skærmen. Årsagen til at tallene skifter hver gang vores tæller runder 127 er at vi fra 127 begynder forfra fra 0. Ender vores talfølge på 126 fortsættes fra 126+6-127-1 = 5. Endes på 125 startes fra 3 osv.

Du kan ændre programmet således at du først lagrer en værdi i " R "-registret (større end 127), og så lader programmet hoppe til loopet. Herved får du konstant tal frem på skærmen større end 128. Dette skyldes jo at vi nu har gjort bit nr. 7 i " R "-reistret 'set' og idet denne bit ikke slettes automatisk vil alle tal være større end 127. Der lægges simpelthen 128 til alle de tal vi så før.

Da du næppe kan lave denne programændring på nuværende tidspunkt så ved tilbage, når du har læst om alle instruktionerne.

Vi skal ved gennemgangen af " IM "-instruktionen omtale " I "-registret mere detaljeret, hvorfor vi springer det over her.

FLAG-EFFEKT:

Det er kun de instruktioner, hvor " A "-registret lagres fra " I " og " R ". Flageffekten er den samme for begge instruktioner:

SIGN : hvis " A "-registrets bit nr. 7 bliver 'set' ved lagringen bliver SIGN-flaget 'set'. Ellers 'reset'.

ZERO : lagres et nul i " A "-registret gøres ZERO-flaget 'set'. Ellers 'reset'.

Dette er ikke de eneste flag, der påvirkes. I appendix er de øvrige flageffekter vist. Specielt PARITETS/OWERFLOW flaget er interessant.

2. LAGRING MELLEM 16-BIT REGISTERPAR:

Der er også mulighed for at foretage indirekte lagring med 16-bit registerpar. Instruktionerne i denne gruppe ser således ud:

LD SP.registerpar

hvor det kun er " HL ", " IX " og " IY " der kan benyttes. Dette betyder at der kun kan være tre instruktioner:

Assembler	Decimal kode
LD SP.HL	249
LD SP.IX	221 249
LD SP.IY	253 249

Som du ser er det kun stakpeger registret, som kan blive lagret med en værdi fra andre registre - med indirekte lagring. Dette betyder jo IKKE at man er afskåret fra det med de øvrige registre i kraft af instruktionerne til indirekte lagring af 8-bit registre. Dette sker imidlertid igen på bekostning af antal bytes instruktionerne fylder samt den tid det tager at udføre instruktionerne. Ønsker vi f.eks. at foretage indirekte lagring af " BC " med " DE " kan dette ikke ske med 16-bit indirekte lagring:

```
LD BC.DE        (findes IKKE)
```

men i stedet bruger vi to instruktioner

```
LD B.D
LD C.E
```

og problemet er løst. Eneste 'bagdel' er et længere tidsforbrug samt en større behov for lagerplads.

Instruktionerne kan benyttes til at genetablere stakken, d.v.s. den anvendt af AMSTRADs operativsystem, eller til at opbygge din egen stak. I forbindelse med opbygningen af din egen stak er det MEGET vigtigt at du sørger for at gemme AMSTRADs stakpeger adresse og stille denne tilbage, når du overgiver operativsystemt styringen af mikroprocessoren. Nedenfor er vist et lille eksempel på hvordan du kan gøre dette:

```
Assembler        Decimal kode

LD HL.ny-stak    33 n n
LD (20000).SP    237 115 32 78
DI               243
LD SP.HL         249
...


...
LD SP.(20000)    237 123 32 78
EI
```

Den viste teknik lagrer AMSTRADs stakpegers adresse i adresse 20000/20001. Adressen hvorfra du ønsker at starte DIN stak lagrer du i 1.instruktion. Du skal imidlertid huske hvordan stakken fungerer. Denne begynder jo oppefra og 'arbejder' sig nedaf - mod mindre adresser. Vi skal senere se på stakken i forbindelse med andre instruktioner eksempelvis " PUSH "," POP " m.fl.

Der er imidlertid flere måder at oprette din egen stak. Stakken du opretter behøver ikke at blive behandlet med samme restriktioner som stakken som AMSTRAD-benytter. Når du benytter AMSTRADs stak må du ALTID sørge for ikke at efterlade tal og adresser etc. på stakken. Dette skyldes jo at AMSTRAD 'regner' med at stakken er intakt, når den skal bruge den. Da der hele tiden ligger vigtige adresser adresser for maskinens funktion på stakken MÅ stakken ikke ændres. Du kan imidlertid lagre på stakken blot du tager disse tal af stakken inden AMSTRAD skal bruge stakken.

Instruktionerne påvirker ikke flagene.

LAGRING MELLEM REGISTER OG AMSTRADS-LAGER

Denne gruppe instruktioner er delt op i instruktioner som lagrer registre med indhold af lageradresser, samt en anden gruppe som lagrer 'den anden vej', d.v.s. kopierer indhold fra registre til lageradresser.

Vi skal i dette afsnit indføre en ny notation, d.v.s. en ny måde at læse instruktionerne på. Du vil i resten af bogen støde på instruktioner, hvor der forekommer paranteser omkring dels registerpar dels adresser. Begge dele er vist nedenfor:

1) parantes om registerpar — LD A.(HL)

2) parantes om adresse — LD A.(adresse)
ex. LD A.(20000)

Når en sådan parantes forekommer betyder det at det er indholdet i den angivne adresse, der skal bruges - og således IKKE eksempelvis det angivne tal, der står i parantesen. I de tilfælde hvor adressen er udskiftet med et registerpar, er det således den adresse, der er lagret i registerparret, hvis indhold der skal bruges. Lad os se på de to eksempler:

```
ad 1)     LD HL.30000
          LD A.(HL)
```

Det er anden instruktion, der er interessant. AMSTRADs mikroprocessor oversætter instruktionen i to omgange: først finder den ud af hvad der er lagret i " HL "-registerparret. Tallet heri er jo et 16-bit tal og kunne derfor være en adresse. Ved anden 'oversættelse' benytter mikroprocessoren tallet fra " HL "-registret som en adresse, hvis indhold hentes fra lageret. Denne lagercelles indhold lagres så endelig i " A "-registret.

```
ad 2)     LD A.(20000)
```

Andet eksempel svarer nøje til første eksempel blot med den forskel at første trin hoppes over. Dette skyldes - meget enkelt - at den adresse, der skal aflæses, den gives direkte i forbindelse med instruktionen, her 20000.

1. LAGRING AF REGISTRE MED INDHOLD AF LAGERADRESSER:

Denne første gruppe giver os nogle yderst nyttige instruktioner, som sætter os i stand til at "aflæse" eller kopiere lageradresser og flytte indholdet fra disse over i registre.

På denne måde kan du f.eks. flytte store dataområder, kopiere dem etc. Det bliver også muligt at bruge AMSTRADs lager som midlertidigt lager for vores maskinkodeprogrammer.

Der findes tre former for eller rettere metoder for at angive adressen fra hvilken indholdet skal hentes. De tre metoder er:

1. DIREKTE ADRESSERING.
2. INDIREKTE ADRESSERING og
3. INDEKSERET ADRESSERING.

Ordet 'adressering' er blot et andet ord for - den måde hvorpå du henter tal fra AMSTRADs lager -. Egentlig består forskellene mellem de forskellige måder i den måde hvorpå du angiver den adresse du vil bruge.

1. DIREKTE ADRESSERING:

Den direkte adressering kaldes således fordi man (direkte) angiver den adresse man ønsker at anvende i forbindelse med instruktionen. Dette i modsætning til at adressen først lagres i et registerpar.

Der findes 7 instruktioner til direkte adressering. Der findes imidlertid kun eet 8-bit register, der kan lagres ved hjælp af direkte adressering. Øvrige instruktioner fortager lagring i registerpar.

Tabel . Direkte lagring.

Assembler	Decimal kode
LD A.(adresse)	58 n n
LD BC.(adresse)	237 75 n n
LD DE.(adresse)	237 91 n n
LD IX.(adresse)	221 42 n n
LD IY.(adresse)	253 42 n n
LD HL.(adresse)	42 n n
LD SP.(adresse)	237 123 n n

Instruktionerne kan godt ved første syn se lidt mærkelige ud, idet der jo på venstre side af punktummet er angivet et registerpar, d.v.s. et 16-bit tal - og på den anden side en adresse i parantes, hvilket jo svarer til et 8-bit tal. Dette er imidlertid ikke tilfældet, idet instruktionerne egentlig skal læses som vist nedenfor:

```
LD BC.(adresse)    =    LD B.(adresse)
                        LD C.(adresse+1)
```

og dermed får vi brug for BEGGE enkeltbyte registre i instruktionen. Kopieringen sker ved at tallet i den angivne adresse kopieres over i low-byte registret. Indholdet i lageradressen lige efter kopieres til highbyte registret.

Det er vigtigt at huske sammenhængen mellem adresse og high/low-register. Lowbyte registret lagres ALTID med indholdet i adressen med det laveste tal.

Dette er selvfølgelig ikke tilfældet for den første instruktion, som kun lagrer " A "-registret.

2. DEN INDIREKTE ADRESSERING:

Den indirekte adressering hedder således, fordi man ad indirekte vej (via et registerpar) angiver, hvilken adresses indhold der skal kopieres over i registeret. Man nævner altså IKKE adressen med tal, men lader dette tal stå i et 16-bit register.

Instruktionerne til indirekte adressering er vist nedenfor. der findes kun instruktioner til lagring af 8-bit registre:

Tabel . Inddirekte lagring.

Assembler	Decimal kode
LD A.(HL)	126
LD A.(BC)	10
LD A.(DE)	170
LD B.(HL)	70
LD C.(HL)	78
LD D.(HL)	86
LD E.(HL)	94
LD H.(HL)	102
LD L.(HL)	97

Instruktionerne læses: kopier indholdet i den adresse, som peges på af registerparret over i det angivne 8-bit register. Der flyttes således KUN een byte fra lager til register i modsætning til den forrige gruppe, hvor der kopieredes to bytes.

Bemærk i hvor høj grad " A "-registret er favoriseret med mulighed for 3 forskellige angivelser af adresse. Dette betyder bl.a. at instruktioner som

LD B.(DE)

ikke findes, men at vi meget let kan 'lave' en sådan. En løsning på problemet kunne være:

```
                assembler       Decimal kode

                LD A.(DE)       170
                LD B.A           71
```

eller

```
                LD H.D           98
                LD L.E          107
                LD B.(HL)        70
```

Indirekte adressering er utrolig meget brugt. Ved at benytte registerparret, som man bruger variable i BASIC, kan instruktionen ofte være del af loops etc., hvorved der ofte genbruges instruktioner. Nedenfor er vist et eksempel på et loop, der er kontrolleret af det tal du lagrer i " B "-registret:

```
                Assembler       Decimal kode

                LD A.0          62 0
                LD HL.adr-1     33 n n
                LD B.(HL)       70
      loop:     INC HL          35
                CP (HL)         190
                JR NC.mindre    48 01
                LD A.(HL)       126
      mindre:   DJNZ loop       16 249
                LD (adr-2).A    50 n n
                RET             201
```

Figur . 'FIND MAKSIMUM' - program.

Flere af instruktionerne er endnu ikke gennemgået og vi skal derfor koncenterere os om instruktionerne der udnytter den indirekte adressering.

Lad os først se på programmets funktion. Hensigten med programmet er at få fundet maksimum af en gruppe tal angivet ved en start- og en slutadresse. Startadressen er i programmet fast - nemlig adr-1 plus een. Antallet af adresser der skal gennemsøges lagres i adr-1. Når den største værdi er fundet lagres denne i adr-2.

Idet " HL "-registret hele tiden bringes til at pege på den adresse, hvis indhold der nu skal testes, kan den indirekte adressering benyttes til at lagre " A "-registret med.

3. INDEKSERET ADRESSERING:

Denne metode er bygget op omkring indeksregistrene, d.v.s. " IX " og " IY ".

At indeksere noget betyder at gøre dette ensbenævnt, og i denne forbindelse - at gå ud fra samme basis. Denne basis er enten en adresse eller en såkaldt forskydning. Indekseringen sker ved at man angiver en basis adresse og at man til denne adresse hele tiden foretager en addition, d.v.s. lægger et tal til adressen, hvorved en ny adresse fremkommer.

Der findes ialt 14 instruktioner - to for hvert 8-bit register.

Tabel . Indekseret lagring.

Assembler	Decimal kode
LD A.(IX+n)	221 115 n
LD B.(IX+n)	221 70 n
LD C.(IX+n)	221 78 n
LD D.(IX+n)	221 86 n
LD E.(IX+n)	221 94 n
LD H.(IX+n)	221 102 n
LD L.(IX+n)	221 110 n

Instruktioner findes også for " IY "-registret.

Tabel . Indekseret lagring.

Assembler	Decimal kode
LD A.(IY+n)	253 115 n
LD B.(IY+n)	253 70 n
LD C.(IY+n)	253 78 n
LD D.(IY+n)	253 86 n
LD E.(IY+n)	253 94 n
LD H.(IY+n)	253 102 n
LD L.(IY+n)	253 110 n

Instruktioner skal læses: Først hentes tallet i registerparret, som jo er et 16-bit tal. Til dette tal lægges det tal du har indsat på 'n's plads. Hermed har du en adresse, hvis indhold kopieres til det 8-bit register du angiver i instruktionen.

Læg mærke til at det er registerparret, der skal bestemme ud fra hvilken adresse indekseringen skal ske. Når du har besluttet dig for 'n'ets, d.v.s. forskydningens størrelse, vil det nemlig kun være indeksregistret du vil kunne ændre indhold i, hvis du skal genbruge instruktionerne. Bemærk at forskydningen maksimalt kan være mellem 0 og 255, idet den kun må 'fylde' 8-bit. Forskydelsen behøver imidlertid IKKE at være positiv. Dette betyder at summen af indeksregistrets indhold og forskydningen kan blive mindre end indholdet i indeksregistret. På næste side er vist hvilke konsekvenser dette kan få for vores muligheder for at adressere lageret.

Heraf ses, hvilke muligheder, der findes til at manipulere (ændre) adressenummeret og dermed indholdet i samme. Er forskydelsen et tal større end 127 vil den adresse, der er lagret i indeksregisterparret, blive reduceret. Dette sker efter følgende regler:

1. n>127: adresse = IX - 256 + n

2. n<128: adresse = IX + n

På IX sted kunne naturligvis lige så godt have stået IY.

Adresse	forskydning	Kommentar
RAMTOP		
...		
...		
40000	127	lageradressen er KUN et eksempel,
39999	126	og kunne være en hvilken som helst
39998	125	anden.
39997	124	
...	...	
		0=< n =< 127 = hop fremad
...	...	
...	...	
39875	2	

```
39874            1
39873            0
39872          255       n               forskydning
39871          254       JR
39870          253
39869          252
 ...           ...
                        128 =< n =< 255 = hop bagud
 ...           ...
 ...           ...
39746          129
39745          128
```

OMBYTNINGS-INSTRUKTIONEN:

I denne gruppe findes 3 instruktioner, som alle kan bytte om på indholdet i de angivne registerpar nævnt i instruktionen. Dette betyder, at indholdet i registrene ikke går tabt, men blot flyttes fra et til et andet registerpar.

De tre instruktioner er:

Assembler	Decimal kode
EX DE.HL	235
EXX	217
EX AF.A'F'	8

Den første instruktion bruges blandt de primære registre. Dette er det eneste registerpar, hvorimellem der findes en sådan instruktion. Instruktionen er meget nyttig, idet der er flere instruktioner som ikke findes for " DE "-registerparret, eller som kræver flere bytes eller længere tid, end den tilsvarende for " HL "-registerparret. Eksempelvis kunne man tænke sig at " HL "-registret indeholdt en lageradresse, hvis indhold skulle sammenlignes med " A "-registret. I dette tilfælde ville man anvende instruktionen:

CP (HL)

hvilken instruktion IKKE findes for " DE "-registret. Indeholder dette register nu adressen ville det være nødvendigt at lagre denne adresse - først i " HL " således at sammenligningen kunne finde sted - dernæst hvis adressen skulle anvendes senere. Anvender du i stedet " EX DE.HL " gemmer du såvel det oprindelige indhold fra " HL " samtidig med at sammenligningen kan ske umiddelbart.

De to andre instruktioner bruges begge på aftryksregistrene. Dette er de to eneste instruktioner, som gør brug af disse registre.

" EXX "-instruktionen er meget omfattende i sin funktion. Den ombytter samtlige registerpars indhold med deres respektive aftryksregistre. Nedenfor er vist hvordan:

FØR:

A	240	A'	140
F	241	F'	141
B	244	B'	144
C	245	C'	145
D	246	D'	146
E	247	E'	147
H	100	H'	200
L	101	L'	201

EFTER:

A	140	A'	240
F	141	F'	241
B	144	B'	244
C	145	C'	245
D	146	D'	246
E	147	E'	247
H	200	H'	100
L	201	L'	101

Den anden instruktion " EX AF.A'F' " medfører kun ombytning mellem " AF "- og dette registerpars aftryksregister.

FØR:

A	240	A'	140
F	241	F'	141

EX AF.A'F'

EFTER:

A	140	A'	240
F	141	F'	241

Exchange-instruktionerne påvirker ikke flagene i " F "-registret. Dette ville også have givet os problemer, idet vi jo i to af instruktionerne netop arbejder på " F "-registret.

I forbindelse med gennemgangen af " stak-instruktionerne " vil du stifte bekendtskab med instruktioner der kan udføre samme funktion som exchange-instruktionerne. Instruktionerne er:

```
PUSH HL
PUSH DE
POP HL
POP DE
```

hvilket ombytter indholdet i " HL " og " DE "-registrene med hinanden.

STAKPEGER - INSTRUKTIONERNE:

Disse instruktioner er delt i to mindre grupper:

1) instruktioner til at lagre tal på stakken samt hente dem tilbage igen.

2) ombytningsinstruktioner forbundet med stakken.

Ombytningsinstruktioner for stakken er medtaget her, idet de mere direkte vedrører instruktionerne i denne gruppe end i den generelle gruppe for ombytningsinstruktioner.

I hele dette afsnit skal vi arbejde med den specielle lagringsmetode, der kaldes STAKKEN. Denne blev beskrevet i forbindelse med gennemgangen af registrene i starten af bogen. Det er vigtigt at du husker denne - så slå tilbage hvis du har glemt.

1. INSTRUKTIONER TIL LAGRING PA/FRA STAKKEN:

Vi skal nu se på de instruktioner, som gør det muligt for os at lagre tal på samt hente tal fra stakken. Instruktionen der lagrer tal på stakken er

PUSH

instruktionen. Sammen med instruktionen angives et registerpar, hvis indhold lagres. Dette sker med low- og highbyte liggende i en bestemt rækkefølge.

Instruktionen der henter tal tilbage fra stakken sker med

POP

som også efterfølges af det registerpar, der skal indeholde tallet der tages fra stakken.

Det er vigtigt at erindre måden hvorpå tal bliver lagret på stakken. Vi kaldte det for LIFO-princippet, som stod for Last In - First Out, d.v.s. sidst ind på stakken skal først ud. Vi beskrev det med en stak, hvor hvert nyt tal lægges nederst i en bunke. Skal vi have fat i et tal på stakken er der kun mulighed for at få fat i den ved at tage tal fra bunden af stakken.

De 6 instruktioner er

Assembler	Decimal kode
PUSH AF	245
PUSH BC	197
PUSH DE	213
PUSH HL	229
PUSH IX	221 229
PUSH IY	253 229

Tabel .Instruktioner til lagring på stakken.

Når stak-instruktionen " PUSH " anvendes sker følgende

1. " SP "-registret bliver først mindsket med een. Dette hænger jo sammen med det faktum at stakken bygges oppefra og nedefter i lageret. " SP " peger konstant på den sidst lagrede byte på stakken. Når dette register nedtælles med een peges således på næste ledige adresse på stakken.

2. Herefter kopieres highbyten fra det angivne registerpar ind i den adresse, som " SP " nu peger på.

3. Sidst mindskes " SP " endnu en gang og lowbyten kopieres ind i denne adresse. " SP " peger nu igen på den sidst anvendte adresse på stakken.

Bemærk at lowbyten lægges SIDST ind på stakken, hvilket betyder at lowbyten kommer til at ligge NEDERST i lageret (i forhold til highbyten). Dette passer netop med den 'normale' måde at lagre dobbeltbyte tal i lageret. Husk f.eks. hvordan

LD HL.(adresse)

lagrer et dobbeltbyte tal fra lageret i " HL "-registret. Her lagres tallet i 'adresse' i lowregistret, d.v.s. " L ".

Den modsvarende instruktion er " POP ", der henter tal tilbage fra stakken. Igen er der 6 instruktioner:

Assembler	Decimal kode
POP AF	241
POP BC	193
POP DE	209
POP HL	225
POP IX	221 225
POP IY	253 225

Tabel .Instruktioner til hentning fra stakken.

Når stak-instruktionen " POP " anvendes sker følgende

1. Først lagres lowbyte registret med indholdet i den adresse som " SP " peger på.

2. Herefter øges " SP " med een og indholdet i den nye adresse kopieres til highbyte registret.

3. Endelig øges " SP " endnu en gang, som så igen peger på den sidst anvendte adresse på stakken.

Stakpegerinstruktioner påvirker IKKE flagregistret.

Selvom man siger at stakken begynder oppefra er dette naturligvis ikke ensbetydende med at den begynder i toppen af maskinens lager. Stakken er en del maskinens dynamiske lager, med hvilken den kan rykke op og/eller ned uden at dette påvirker indholdet på stakken.

Stakken kommer i anvendelse, når du bruger bl.a.

GOSUB linienr.

ON ERROR GOTO

ON BREAK GOTO

kommandoerne benyttes stakken til at lagre linienumrene, hvortil maskinen skal vende tilbage efter subrutinen eller 'trap'-rutinen.

Vi skal nu se lidt på anvendelsen af stakinstruktionerne " PUSH " og " POP ". Læg mærke til at det i reglen er ligegyldigt, hvilket registerpar, der er lagret på stakken, når man senere skal tage indholdet af stakken igen. Det er altså ikke således at hvis " BC " registerparrets indhold bliver lagret på stakken, så skal dette tal tages af stakken tilbage til " BC ". Dette betyder at vi kan benytte stakinstruktionerne til at 'bytte om' på indholdet i to registerpar. Dette er vist nedenfor, hvor vi ombytter indholdet i " DE " med " BC ":

Assembler	Decimal kode
PUSH BC	197
PUSH DE	213
POP BC	193
POP DE	209

Tænker vi os at " BC "-registret indeholder tallet 30000, indeholder " B " 117 og " C " 48. Indeholder " DE " 45034 er " D " 175 og " E " 234. Vi kan så vise hvordan registrenes indhold skifter samt indholdet på stakken, idet vi forestiller os at stakken er nået 'ned' til lageradresse 30000:

```
SP = 30000                          (30000) = ?????

                    PUSH BC
SP = 29998                          (29999) = 48  : B
                                    (29998) = 117 : C

                    PUSH DE
SP = 29996                          (29997) = 234 : D
                                    (29996) = 175 : E

                    POP BC
SP = 29998                          (29996) = 175 : C
                                    (29998) = 234 : B

                    PUSH DE
SP = 30000                          (29997) = 117 : D
                                    (29996) = 48  : E
```

Bemærk hvordan " SP "-registret mindskes for hver " PUSH "-instruktion der udføres og øges med " POP ". Hvis vi efterfulgte rigtig mange " PUSH "-instruktioner efter hinanden ville det være muligt at " SP " tilsidst ville blive nul (og negativ). Denne situation ville medføre at AMSTRAD ville 'crashe'. Dette skyldes at nødvendige data og instruktioner ville blive overskrevet. Dette er en af grundene til at du skal bruge de to instruktioner SYMMETRISK, d.v.s. antallet af " PUSH " og " POP " skal være lig hinanden. Dette er imidlertid ikke en generel regel, hvilket vi senere skal se på hvorfor.

Du kan 'simulere' ovennævnte med et BASIC-program, som ser således ud:

```
Linienr.  Kommando

    10    REM***** GOSUB TEST *******
    20    GOSUB 30
    30    GOTO 20
    40    REM************************
```

I løbet af et ganske kort øjeblik vender AMSTRAD tilbage med fejlmeddelelsen " MEMORY FULL ". Dette betyder netop at linienummeret 30, som er den linie som skal hoppes tilbage til ved en eventuelt RETURN-kommando, har 'fyldt' lageret op.

Vi kan med det næste program finde ud af hvormange 'linienumre' der egentlig lagres i maskinen før fejlmeddelelsen kommer:

```
Linienr.  Kommando

    10    REM***** GOSUB TEST *******
    15    a=0
    20    GOSUB 30
    30    a=a+1
    40    PRINT a
    50    GOTO 20
    40    REM************************
```

Når du udfører det ovenfor viste program skulle du kunne se at a opnår værdien 83 før fejlen opstår. Du kan eventuelt teste om dette tal har noget at gøre med antallet af bytes, d.v.s. den lagerplads, der er tilrådighed for dit BASIC-program. Dette sker jo med

```
MEMORY adresse
```

kommandoen, hvor adresse er den nye overgrænse for BASIC-området.

Arsagen til at vi ikke får denne fejl skyldes at vi jo anvender RETURN-kommandoen. Denne kommando trækker et linienummer ned fra stakken og sørger for at AMSTRAD hopper til denne. Herfra fortsætter udførelsen af programmet. Det vi egentlig gør er at vi igen ØGER " SP "-registret, hvorved lageret IKKE fyldes op med linienumre.

En anden meget vigtig funktion, som du kan anvende stakken til er at styre AMSTRAD til en bestemt adresse, hvor den vil fortsætte udføringen af instruktioner. Som vi senere skal se i forbindelse med " RET "-instruktionen tages de to sidst indlagte tal fra stakken af og læses ind i Program Tælleren (P.C.). Dette register peger jo altid på den næste adresse, hvis indhold der skal opfattes som den næste instruktion der skal udføres. Vi kan med " PUSH " instruktionen lægge den adresse vi ønsker at hoppe til op på stakken og så få udført en " RET "-instruktion. Herved vil AMSTRAD hoppe til den adresse vi har lagt på stakken. Her er det selvfølgelig ekstra vigtigt at rækkefølgen af low-high bytene ligger i korrekt rækkefølge. Sker dette ikke vil AMSTRAD hoppe til den forkerte adresse, og muligheden for at maskinen 'crasher' er dermed stor. Dette skyldes jo at vi ikke aner noget om hvad det er for instruktioner som AMSTRAD skal udføre på den pågældende adresse.

Princippet er anvendt nedenfor, men når du indtaster RUN til programmet skal du sørge for ikke at have noget vigtigt program i maskinens lager.

```
                Assembler       Decimal kode

30000:          LD HL.30005     33 53 117
                PUSH HL         229
                RET             201
30005:          LD HL.30000     33 48 117
                PUSH HL         229
                RET             201
```

BASIC-loaderen er vist nedenfor:

```
Linienr.  Kommando

    10    FOR a=1 TO 10
    20    READ f
    30    POKE a+42999,f
    40    NEXT a
    50    DATA 33,53,117,229,201,33,48,117,
          229,201
```

Figur . 'Illustration' af PUSH-instruktionen.

Prøv så at få programmet udført !.

Bliv ikke overrasket over at du ikke kan få AMSTRAD til at 'høre' dine indtastninger EFTER at du har startet programmet. Årsagen til dette er at vi har startet 'et uendeligt loop'. Hver gang vi skubber (PUSH) en adresse op på stakken og udfører en " RET "-instruktionen, så er stakken egentlig tømt. Dette hænger sammen med at " RET " øger " SP "-registret nøjagtig som " POP ". Loopet består så i at AMSTRAD først hopper til 30005, så til 30000, så tilbage til 30005, osv.

Da dette loop udfører hele tiden 'vender' AMSTRAD ikke tilbage til den rutine i det operativsystem, der udskriver de indtastede symboler på skærmen. Dette betyder at AMSTRAD egentlig opfatter dine indtastninger, d.v.s. de lagres i en inputbuffer, men de vises ikke på skærmen. Inputbufferen ligger fra adresse 368 og 256 adresser frem. Denne fyldes med de koder du indtaster, hvilket sker via operativsystemet. Koderne du finder i dette område er imidlertid 'behandlet' af operativsystemet, hvilket bl.a. betyder at alle kommandoer erstattes af den interne kode for kommandoen - i stedet for at lagre bogstaverne i kommandoerne.

Inputbufferens størrelse (256 bytes=256 bogstaver) kan testes ved at holde en tast nede konstant. Hermed repeteres bogstavet i bufferen samt på skærmen. Efter 256 viste bogstaver begynder AMSTRAD at 'råbe' op og stopper derefter for indtastningen af flere. Bufferen er fyldt op - 256 bogstaver.

Det er ikke umiddelbart muligt at teste dette fra et BASIC-program, idet den linie, der udføres konstant bliver lagret i denne inputbuffer. Dermed ændres indholdet i bufferen hele tiden. I stedet kan vi fra et maskinkodeprogram flytte indholdet af adresse 64-318 til et andet sted i lageret, hvor vi fred og ro kan se på de lagrede koder.

Selve dit BASIC-program ligger lagret i maskinen fra adresse 368 og opad i lageret. Vi kan forvisse os om dette ved at aflæse de koder der ligger i oprådet 368 og opefter. Dette sker bedst ved at tage en kopi af området og flytte denne op i lageret.

Nedenfor er vist et program, der kan foretage den nævnte kopiering at BASIC-området:

```
Assembler       Decimal kode

DI              243
LD HL.368       33 112 1
LD DE.20000     17 32 78
LD BC.256       1 0 1
LDIR            237 176
EI              251
RET             201
```

Figur . Kopiering af indhold i - BASIC område -.

BASIC-loaderen er vist nedenfor:

```
Linienr.  Kommando

    10    FOR a=1 TO 14
    20    READ f
    30    POKE a+42999,f
    40    NEXT a
    50    DATA 243,33,112,1,17,32,78,1,0,1,
          237,176,251,201
```

Figur .Program til kopiering af inputbuffer.

Til analyse af det kopierede BASIC-område laver du blot et program, der " PEEK "er de adresser som bufferen er kopieret til. Hermed kan du få megen information om hvordan AMSTRAD lagrer BASIC-linier etc.

```
          Kommando

(først)   CALL 43000

          FOR z=20000 TO 20256:PRINT z;PEEK(z):NEXT z
```

(hold et skærmbillede fast med <ESC>-tasten. Tryk på en anden tast for næste skærmbilled>

Med det ovenfor viste program fås følgende koder frem på skærmen: (adresserne er taget fra)

	001	002	003	004	005	006	007	008	009	010	011	012	013	014	015
1	18	0	10	0	158	32	13	5	0	225	239	15	32	236	32
2	25	14	0	14	0	11	0	20	0	195	32	13	14	0	230
3	0	23	0	30	0	190	32	13	5	0	225	244	31	0	0
4	247	39	144	44	13	14	0	230	0	11	0	40	0	176	32
5	13	5	0	225	0	50	0	50	0	140	32	50	52	51	44
6	51	51	44	49	49	50	44	49	44	49	55	44	51	50	44
7	55	56	44	49	44	48	44	49	44	50	51	55	44	49	55
8	54	44	50	53	49	44	50	48	49	0	0	0	0	0	193
9	4	0	0	0	112	132	0	0	198	4	0	00	73	136	0 ....

Hver BASIC-linie fylder et vist antal bytes i lageret. Dette antal bytes svarer til de to første tal for enhver BASIC-programlinie. Første eksempel ses i (001,1) og (002,1). Første tal er lowbyten, andet high byten. Linie fylder således 18 bytes og dette regnet eksklusiv de første to bytes. Tæl derfor 18 bytes frem fra (003,1) - du skulle ende på (005,2) udfor nullet. De bytes du talte over udgør første BASIC-linie omsat fra bogstaver til koder.

Efter antal bytes linie fylder kommer endnu to der er linienummeret, igen med low først. Første linienummer er således 10. Tilsvarende kan du se at efter de to første koder for anden linie, d.v.s. (006,2) og (007,2) er andet linienummer 20. Dette ses af (008,2) og (009,2). Tilsvarende for hver efterfølgende linie.

Efter linienummeret starter de egentlige kommandoer som linie udgør. HUSK at programmet, der ligger i lageret er det ovenfor viste program til kopiering af BASIC-området. Du kan derfor sammenholde dette program med koderne efterhånden som vi kommer frem.

ARITMETIK-INSTRUKTIONERNE:

At kunne læse og skrive bytes fra og til lageret har som vi har set en vigtig betydning. Dette er bare ikke tilstrækkeligt, hvis vi skal kunne udføre noget nyttigt, som f.eks. at lægge indholdet i forskellige registre sammen, gange dem med hinanden osv. Her får vi brug for at kunne `regne` med computerens registre. Dette er hvad vi skal beskæftige os med i de næste par afsnit. Instruktionerne hertil består egentlig kun af additions- og subtraktionsinstruktioner, samt nogle specialudgaver heraf. Men vi skal dog se hvorledes disse kan kombineres til såvel multiplikation som division.

Aritmetikinstruktionerne er gennemgået i to større grupper og der er afslutningsvis gennemgået sammenligningsinstruktionen, som ligner subtraktions instruktionen meget.

De tre grupper bliver herved:

1. ADDITIONS-INSTRUKTIONERNE
2. SUBTRAKTIONS-INSTRUKTIONERNE
3. SAMMENLIGNINGS-INSTRUKTIONERNE

ADDITIONS-INSTRUKTIONERNE:

I denne første gruppe er der tre forskellige instruktioner, som hver har sit afsnit:

1. " ADD " -instruktionen,
2. " ADC " -instruktionen og
3. " INC " -instruktionen

som alle anvendes på såvel enkelt- (8 bit) som dobbeltbyte tal (16 bit).

1. " ADD " -instruktionen:

ADD betyder -adder- eller -læg sammen-. Der er mulighed for at lægge indholdet i et register sammen med et andet, lægge det til indholdet i en lageradresse og endelig at lægge det sammen med en direkte angiven konstant. Vi skal se på alle tre muligheder som ved kombination giver os endnu flere muligheder.

A) Addition af enkelt bytes (8 bit + 8 bit):

ADD betyder som sagt summer eller -læg til-. Med instruktionen

```
ADD n
```

kan du addere et tal til indholdet i " A "-registret. Du udskifter 'n' med det tal du skal addere til, men husk at 'n' nødvendigvis må være mellem 0 og 255 (-127 og 128).

Med

```
ADD r
```

kan du addere indholdet i et register til indholdet i " A "-registret. Her udskifter du 'r' med det register, hvis indhold du vil addere. 'r' kan således være A,B,C,D,E,F,H eller L. Ønsker vi f.eks. at addere 8 til indholdet i " A "-registret kunne dette ske med:

```
LD B.8
ADD B
```

Dette er imidlertid et meget dårligt eksempel, idet vi lige så godt kunne anvende den direkte addition med " ADD n ". Men i det tilfælde, hvor det tal der skal adderes varierer kan " ADD r " bedst anvendes.

Den sidste variation er instruktionen

ADD (r)

hvor 'r' igen står for indholdet, men i et registerpar. Læsningen af instruktionen kan bedst forståes med et eksempel som det vist nedenfor:

ADD (HL)

Her indeholder " HL "-registret et tal mellem 0 og 65535. Dette tal skal i forbindelse med instruktionen opfattes som en adresse, nemlig adressen på en lagercelle. Parantesen omkring registernavnet betyder at det er indholdet i den adresse som registeret peger på, der skal bruges. Det er således IKKE indholdet i " HL ", der skal bruges. Computeren henter altså indholdet i " HL ", tænker på dette tal, som en adresse i sit lager og henter indholdet i lageradressen. Tallet der står i lageradressen adderes så endelig indholdet i " A "-registret.

Du skal ikke lade dig forvirre af at der kun angives eet register i forbindelse med instruktionen. Dette hænger jo sammen med at man kun kan addere til " A "-registret.

Nedenfor er samtlige instruktioner i denne gruppe vist. Som du kan se viser instruktionerne kun den ene part. Den anden er ALTID " A "-registret, d.v.s. når der står " ADD 34 " skal det læses : " læg tallet 34 til indholdet i A-registret og skriv resultatet i A-registret ".

Assembler	Decimal kode
ADD B	128
ADD C	129
ADD D	130
ADD E	131
ADD H	132
ADD L	133
ADD (HL)	134
ADD A	135
ADD (IX+n)	221 134 n
ADD (IY+n)	253 134 n
ADD n	230 n

Tabel 1. Oversigt over " ADD "-instruktionerne.

Instruktionerne kan, som det er vist i figuren nedenfor benyttes til at lægge to 8 bit tal sammen. Større tal kan imidlertid også klares ved at kombinere de nævnte instruktionerne. En enkelt instruktion kan imidlertid ikke, idet ingen af addenderne kan blive større end 255, idet dette kræver samtlige bit "set":

bit nr.	7	6	5	4	3	2	1	0
	1	1	1	1	1	1	1	1

Men når nu to tal, i størrelsen 0 til 255 (-127 til 128) lægges sammen kan summen jo komme op på maksimalt 510. Idet " A "-registret jo kun er et 8-bit register kan man undre sig over hvordan registret kan vise et så stort resultat. Her kommer flagene igen ind og redder computeren. Som vi omtalte ved gennemgangen af " F "-registret er CARRY-flaget er slags 'menteflag'. Dette betyder, at hvis resultatet overstiger de 255 som " A "-registret egentlig kun kan indeholde, bliver CARRY-flaget "set".

Flag-effekten ved " ADD "-instruktionen er vist nedenfor, idet effekten er ens for samtlige instruktioner:

FLAGEFFEKT:

SIGN: -påvirkes af resultatets fortegn. Er resultatet positivt eller nul bliver SIGN-flaget "reset". Ellers bliver det "set".

ZERO: -påvirkes af resultatets størrelse. Er resultatet nul bliver ZERO-flaget "set". Ellers "reset".

CARRY: -påvirkes af resultatets størrelse. Bliver resultatet større end 255 bliver CARRY-flaget "set". Ellers bliver flaget "reset".

SUBTRAKTION: -bliver "reset".

De øvrige flags påvirkning er vist i Appendix bagest i bogen, og er ikke medtaget her.

Vi skal se lidt nærmere på flag-effekternes betydning som signaler til os om resultatets størrelse. Nedenfor er vist to eksempler der skulle vise dig hvor vigtig flagene er.

Ex. 1:

Assembler	Decimal kode
LD A.125	62 125
LD H.34	38 34
ADD A.H	132
LD (40000).A	50 64 156
RET	

Ex. 2:

Assembler	Decimal kode
LD A.125	62 125
LD H.134	38 134
ADD A.H	132
LD (40000).A	50 64 156
RET	

BASIC-loaderen er vist nedenfor. Programmet udfører endvidere de to maskinkode programmer.

```
linienr.  kommando

   100  c=42999
   200  FOR a=1 TO nn
   300  READ f
   400  POKE a+c,f
   500  NEXT a
   600  DATA 62,125,38,34,132,50,n,n,201
   700  DATA 62,125,38,134,132,50,n,n,201
   800  CALL 43000
   900  PRINT PEEK (40000)
  1000  CALL 43009
  1010  PRINT PEEK (40000)
```

Når du har indtastet og fået udført BASIC-programmet skulle tallene 159 og 3 gerne stå på Monitoren. Summen af 125 og 34 passer udemærket med det første tal på skærmen.

Det andet eksempel er lidt problematisk: 125 + 134 = 3 ?. Dette skyldes naturligvis at resultatet overstiger 255, nemlig ialt 259. Som nævnt i forbindelse med flageffekterne bliver CARRY-flaget "set", når resultatet overstiger 255. Vi kan således gå ud fra at dette flag er "set". Nedenfor er vist hvordan man kunne kontrollere dette.

```
Assembler        Decimal kode

LD A.x           62 x
LD H.y           38 y
ADD A.H          132
LD (40000).A     50 N N
RET NC           208
LD A.1           62 1
LD (40001).A     50 N N
RET              201
```

Eneste forskel fra de to første eksempler er tilføjelsen af de 4 sidste linier. Tilføj disse til een af DATA linierne i den sidst viste BASIC-loader og ændr så 400 til:

```
400  PRINT PEEK (40000)+256*PEEK (40001)
```

Vi benytter således endnu en lageradresse til at angive resultatet. Hermed vil vi altid kunne få det korrekte resultat.

Med denne gruppe af instruktioner kan vi lave programmer til multiplikation af enkelt- og dobbeltbyte. Nedenfor skal vi se på et program til multiplikation af enkeltbytes. Programmet er opbygget efter den måde hvorpå man normalt beregner et produkt af to tal, eksempelvis 5 gange 6. Vi siger netop " fem gange seks ", hvilket betyder at vi blot skal simulere dette i vores program. Hvordan nu det ?. Jo først nulstiller (clearer) vi et register. Herefter benytter vi et andet register med tallet fem i som tæller. Dette gør vi fordi vi jo skal lægge seks sammen med sig selv fem gange. Endelig skal vi bruge et sidste register eller en lageradresse til at lagre resultatet. Dette kunne se således ud:

```
                Assembler           Decimal kode

                LD HL.40000         33 64 156
                LD A.(HL)           126
                LD B.A              71
                INC HL              35
                LD A.(HL)           126
        loop:   DEC B               5
                LD (40002).A        50 n n
                RET Z               200
                ADD (HL)            134
                JR loop             24 248
```

Figur . Program til multiplikation af to 8-bit tal.

Lagringen af maskinkode programmet sker i nedenstående BASIC-program, som også varetager testningen af programmet:

```
Linienr.  Kommando

    5     REM****** LAGRING AF PROGRAM *******
   10     FOR a=1 TO 15
   20     READ f
   30     POKE a+42999,f
   40     NEXT a
   50     DATA 33,64,156,126,71,35,126,5,
          50,66,156,200,134,24,248
   60     REM****** TEST AF MULTIPLIKATION ******
   70     INPUT "indtast første tal ";a
   80     INPUT "indtast andet tal ";b
   90     POKE 40000,a:POKE 40001,b
  100     CALL 43000
  110     PRINT PEEK (40002)
  120     GOTO 60
```

Figur . BASIC-program til lagring samt test af enkelt byte multiplikation.

Som nævnt benytter vi et tælleregister, som er vores kontrol red antallet af gange vi adderer til resultatregistret. tælleregistret er " B "-registret, som lagres med det ene af de tal vi POKE'r ned i lageret via BASIC-programmet, nemlig det første tal vi angiver (via adresse 40000). Det andet tal vi indtaster er det tal vi skal summere til " A "-registret " B " gange. Årsagen til anvendelsen af " HL "-registret skyldes alene en let lagring af såvel " A "- som " B "-registret.

I programmet nedtæller vi " B "-registret og lagrer straks efter resultatet (vi er nået til) i adresse 40002. Herefter testes ZERO-flaget for om nedtællingen af " B "-registret skulle have resulteret i at " B " blev nul. Vi slutter når dette sker.

Bliver " B " IKKE nul ved nedtællingen summeres endnu en gang hvilket fortsætter til vi har summeret " B "-gange.

Når du tester programmet skal du blot indtaste forskellige værdier til de to INPUT-kommandoer. Prøv tal der giver resultater såvel større som mindre end 255. Bemærk dog at du IKKE kan indtaste tal større end 255 til INPUT-kommandoerne. Dette skyldes at tallene umiddelbart efter skal lagres i computerens lager med POKE-kommandoen. Da hver lagercelle maksimalt indeholder 8-bit og dermed et tal på højest 255 vil computerens kontrolsystem syntax-kontrol) standse dig. Dette har altså intet med vores maskinkodeprogram at gøre.

Vi skal senere se på en mere effektiv måde at multiplicere tal med hinanden på. Denne teknik benytter instruktioner der kan rykke bits i et register. Herom senere i afsnittet om ROTATIONER OG SKIFTS.

B: Addition af dobbelt byte tal:

Disse instruktioner kan bruges mellem registerpar, hvorved vi får mulighed for at lægge 16 bit tal sammen. 16-bit tal er tal i størrelsen 0 til 65535 (eller -32767 til 32768).

De 12 instruktioner i denne gruppe er vist nedenfor:

Assembler	Decimal kode
ADD HL.BC	9
ADD HL.DE	25
ADD HL.HL	41
ADD HL.SP	57
ADD IX.BC	221 9
ADD IX.n	221 25 n
ADD IX.IX	221 41
ADD IX.SP	221 57
ADD IY.BC	253 9
ADD IY.n	253 25 n
ADD IY.IY	253 41
ADD IY.SP	253 57

Figur . " ADD "-instruktionerne for registerpar.

Når disse instruktioner benyttes foretages additionen i to dele, hvor 'low' bytene adderes først, eksempelvis hvis det var:

ADD IX.BC

blev indholdet i " X " og " C "-registrene summeret først. Hvis der er overflow, d.v.s. at summen fylder mere end 8-bit, overføres denne CARRY (mente) til addition af 'highbytene'. Når additionen af lowbytene er sket summeres 'highbytene', eller svarende til vores eksempel: registrene " I " og " B " summeres.

Bemærk at summen ALTID kommer til at stå i registerparret der står nærmest " ADD "-ordet.

Som med enkelt byte " ADD "-instruktionen er der grænser for hvor store tal et registerpar kan indeholde. Et registerpar udgør jo ialt 16 bit, som hvis alle var "set":

```
bit nr.  15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00

          1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
```

ville svare til tallet 65535 (eller -32767). Dette betyder at når summen overstiger dette tal må vi igen tage CARRY-flaget med i vores beregninger af resultatet. Her betyder det at CARRY-flaget er "set" bare IKKE at vi skal lægge 256 til resultatet. Her angiver CARRY at der skal lægges 65536 til resultatet. Lav selv et eksempel som kan illustrere dette - efter samme opskrift som det vist under 8-bit " ADD ".

Problemet fra adsnittet om addition af to 8-bit tal kan vi nu løse væsentlig lettere, idet vi ikke behøver at overvåge CARRY-flaget ved additionen:

```
Assembler        Decimal kode

LD HL.125        33 125 0
LD BC.134        1 134 0
ADD HL.BC        9
LD (40000).HL    34 n n
RET              201
```

Bemærk her at lagringen af " HL "-registret sker som omtalt under registerpar lagring. Lowbyte-registret, her " L ", lagres i den angivne adresse, mens highbyte-registret, " H ", lagres i adressen umiddelbart over.

FLAGEFFEKT:

Kun CARRY-flaget påvirkes af " ADD "-instruktionen for addition med registerpar. CARRY-flaget bliver "set", når der forekommer binært overflow, hvilket betyder at resultatet af additionen ikke kan 'stå' i registerparret. Sagt med andre ord skal resultatet være større end 65535 (eller regnet med fortegn 32767). CARRY-flaget vil blive "reset", hvis overflow ikke forekommer.

Helt korrekt er det imidlertid ikke at det kun er CARRY-flaget, der påvirkes, idet SUBTRAKTIONs-flaget altid gøres "reset".

" ADC "-INSTRUKTIONEN:

Denne instruktions gruppe udfører " ADD ", men tager hensyn til CARRY-flaget. Dette sker således at hvis CARRY-flaget er "set" INDEN " ADC "-instruktionen udføres vil der blive medtaget en mente i additionen. Instruktionen kan således løse en del problemer som vi var inde på i forrige afsnit.

Instruktionen findes for addition af såvel enkelt- som dobbelt byte tal.

Addition med CARRY for enkelt byte tal:

I denne gruppe er der 11 instruktioner omfattende de nedenfor viste:

Assembler	Decimal kode
ADC B	136
ADC C	137
ADC D	138
ADC E	139
ADC H	140
ADC L	141
ADC (HL)	142
ADC A	143
ADC (IX+n)	221 142 n
ADC (IY+n)	253 142 n
ADC n	206 n

Tabel 1. Oversigt over " ADC "-instruktionerne.

Instruktionerne kan, som det er vist i foregående afsnit benyttes til at lægge to 8 bit tal sammen. Større tal kan imidlertid også klares ved at kombinere de nævnte instruktionerne. En enkelt instruktion kan imidlertid ikke, idet ingen af addenderne kan blive større end 255, idet dette kræver samtlige bit "set".

Vi kan illustrere " ADC "-instruktionen i funktion ved at foretage to additioner, hvor den første gør CARRY-flaget "set", hvilket så benyttes i den anden addition:

```
Assembler        Decimal kode

LD A.150         62 150
ADD A            135
ADC A            143
LD (40000).A     50 64 156
RET
```

Figur . Illustration af " ADC "-instruktionen.

Lav selv BASIC-programmet til lagring af maskinkoden. Udfør programmet med CALL-kommandoen som sædvanligt, og du skulle se tallet 89 vise sig på skærmen, når du har POKEt adresse 40000.

I maskinkode programmet adderer vi først 150 med sig selv, d.v.s. 150 + 150 = 300. Dette resultat lagres i " A "-registret, men idet tallet er større end 255 - fratrækkes 256 fra resultatet hvorved resten på 44 lagres i " A "-registret. For at vise at dette er sket gøres CARRY-flaget "set". Bemærk at denne første addition skete med " ADD "-instruktionen.

Anden addition sker med " ADC ". Her adderes således " A " igen med sig selv. " A " indeholder nu kun de 44 og summen bliver derfor 88 - og der sker ingen CARRY-flag påvirkning denne gang. Hvad er nu årsagen til at vi ser et resultat der er netop een større end de 88 ? - Netop....

De 88 forøges med een, idet CARRY-flaget var "set" INDEN " ADC "-instruktionen.

Du kan med rette siges at eksemplet ikke er særlig godt - i hvert tilfælde til at fordoble indholdet i " A "-registret to gange. Det var nu heller ikke hensigten, men det kunne jo blive din opgave at finde ud af.

Addition med CARRY for registerpar:

Denne gruppe består kun af 4 instruktioner, idet der ikke findes instruktioner til indeksregistrene " IX " og " IY ". Instruktionerne er vist nedenfor:

Assembler	Decimal kode
ADC HL.BC	237 74
ADC HL.DE	237 90
ADC HL.HL	237 106
ADC HL.SP	237 122

Figur . " ADC "-instruktionerne for registerpar.

Når disse instruktioner benyttes foretages additionen i to dele, hvor 'low' bytene adderes først, eksempelvis hvis det var:

ADC HL.BC

blev indholdet i " L " og " C "-registrene summeret først. Hvis der er overflow, d.v.s. at summen fylder mere end 8-bit, overføres denne CARRY (mente) til addition af 'highbytene'. Når additionen af lowbytene er sket summeres 'highbytene', eller svarende til vores eksempel: registrene " H " og " B " summeres.

Summen kommer altid til at stå i " HL "-registret. Denne instruktionsgruppe er således med at at gøre " HL "-registerparret til vel det mest fleksible af dem alle.

Som med enkelt byte " ADC "-instruktionen er der grænser for hvor store tal et registerpar kan indeholde. Dette omtalte vi også i forbindelse med " ADD "-instruktionen.

Igen er det kun CARRY- og SUBTRAKTIONS flagene der påvirkes.

Nedenfor skal vi se hvordan vi kan lave en 16-bit addition med " ADC "-instruktionen:

Assembler	Decimal kode
LD A.(adresse 11)	58 n n
LD HL.adresse 21	33 n n
ADD A.(HL)	134
LD (adresse 31).A	50 n n
LD A.(adresse 1h)	58 n n
DEC HL	43
ADC A.(HL)	142
LD (adresse 3h).A	50 n n

svarende til at:

adresse 11, adresse 21 og adresse 31

indeholder lowbytene.

adresse 1h, adresse 2h og adresse 3h

indeholder highbytene.

Adresse 1 og 2 indeholder lowbytene af tallene der skal adderes, mens resultatet står i adresse 3:

```
lageradresser:      ....
                    ....
                    adresse 1h
                    adresse 1l
                    ....

                    ....
                    adresse 2h
                    adresse 2l
                    ....

                    ....
                    adresse 3h
                    adresse 3l
                    .....
```

På denne måde kan vi placere resultatet i den lagercelle, der bedst passer os. Denne form for addition kan imidlertid syntes noget mere omstændelig end den nedenfor viste:

Assembler	Decimal kode
LD HL.(adresse 1l)	42 n n
LD BC.(adresse 2l)	237 75 n n
ADD HL.BC	9
LD (adresse 3l).HL	34 n n

som ikke giver os problemer med CARRY-flag etc.

Skulle vi ønske at addere tal af størrelsen op til 4.3 mill skulle vi bruge et program til 32-bit addition. Et sådant er vist nedenfor, hvor de to tal der skal summeres samt resultatet er placeret som vist nedenfor:

```
lageradresser:      ....
                    ....
                    adresse 1-3
                    adresse 1-2
                    adresse 1-1
                    adresse 1-0
                    ....

                    ....
                    adresse 2-3
                    adresse 2-2
                    adresse 2-1
                    adresse 2-0
                    ....

                    ....
                    adresse 3-3
                    adresse 3-2
                    adresse 3-1
                    adresse 3-0
                    .....
```

Man skal imidlertid være MEGET påpasselig med rækkefølgen af de fire dele af hvert tal. Første lagercelle, d.v.s. alle der ender med "-0" indeholder antallet af '1-ere'; anden celle indeholder antallet af '256-ere'; tredie antallet af '65536-ere' osv.

Selve programmet er egentlig meget simpelt:

Assembler	Decimal kode
LD HL.(adresse 1-0)	42 n n
LD BC.(adresse 2-0)	237 75 n n
ADD HL.BC	9
LD (adresse 3-0).HL	34 n n
LD HL.(adresse 1-2)	42 n n
LD BC.(adresse 2-2)	237 75 n n
ADC HL.BC	237 74
LD (adresse 3-2).HL	34 n n

Instruktionen " ADC " er meget vigtig i det tilfælde at CARRY-flaget skulle blive "set" af første addition.

" INC "-INSTRUKTIONEN:

INC er forkortelsen for det engelske ord INCREMENT, som kan oversættes til -forøgelse-. Hermed kan man betragte " INC "-instruktionen som en specialversion af " ADD "- og " ADC "-instruktionerne. " INC " adderer imidlertid kun een til indholdet af det angivne register eller lageradresse.

For denne gruppe findes også instruktioner for såvel enkelt som dobbelt bytes.

A. " INC " for enkeltbyte registre og lageradresser:

Denne instruktion findes for samtlige 'normale' registre, ialt 10 instruktioner:

Assembler	Decimal kode
INC B	4
INC C	12
INC D	17
INC E	28
INC H	34
INC L	44
INC (HL)	52
INC A	60
INC (IX+n)	221 52 n
INC (IY+n)	253 52 n

Tabel 1. Oversigt over " INC "-instruktionerne.

" INC " forøger som nævnt indholdet i det angivne register eller indhold i lageradressen med een. Instruktionen er derfor hyppigt anvendt som tæller i hvilke man har brug for enkeltvis forøgelse.

Vi kan illustrere brugen af tællere med nedenstående program, som kan printe samtlige ASCII-karakterer op på skærmen.

```
        Assembler       Decimal kode

        DI              243
        LD BC.32649     1 137 127
        OUT (C).C       237 73
        LD C.32         14 32
        LD A.C          121
        CALL 5120       205 n n
loop:   LD A.C          121
        CP 255          254 255
        JR C end        56 3
        INC C           12
        JR loop         24 n
 end:   EI              251
        RET             201
```

Figur . Illustration af " INC "-instruktionen til printning af ASCII symboler m.m.

Programmet udfører det samme som nedenstående BASIC-program.

```
linienr. kommando

    10 FOR a=32 TO 255
    20 PRINT CHR$(a);
    30 NEXT a
```

og ved at sammenligne de to programmer skulle du kunne se hvordan maskinkode programmet fungerer.

Idet vi skal anvende en af udskrivningsrutiner i computerens operativsystem skal dette først 'pages' ind. Dette sker som sædvanlig med de tre første instruktioner.

Da udskrivningen af koder mellem 0 og 31 ikke forårsager symboler på skærmen er disse undladt. Flere af disse kan foretage ændringer i PEN og PAPER, slette skærmen etc.

Vi starter derfor med at lagre kode 32 i " C "-registret. Dette bliver således den første værdi, der skal udskrives et symbol for. Det er så meningen at " C "-registret skal forøges med " INC "-instruktionen for hele tiden at kunne indeholde den decimale værdi for det næste symbol, der skal udskrives.

Vi må dog på en eller anden måde få stoppet denne udskrivning, idet vi ved forøgelse af tallet 255 i " C "-registret kommer tilbage til nul:

```
bit nr.  7 6 5 4 3 2 1 0
         1 1 1 1 1 1 1 1
    +1   0 0 0 0 0 0 0 1

    (1)  0 0 0 0 0 0 0 0
```

hvilket jo svarer til nul, idet CARRY ikke medtælles ved " INC ".

Det er imidlertid netop dette at CARRY bliver "set", idet vi når een over 255, som vi benytter som signal til at programmet skal standse. Derfor instruktionerne

```
        JR C end
        ...
        ...
end:    EI
        RET
```

som vi alle sidenhen skal komme tilbage til. Bemærk blot at " JR C "-instruktionen tester om CARRY-flaget er "set" (derfor det store C) og hopper i givet fald til " EI "-instruktionen.

Bliver CARRY ikke "set" i forbindelse med forøgelsen af " C "-registret hoppes tilbage til instruktionen udfor 'loop:'. Herfra fortsættes udførelsen af programmet.

Inden vi forlader programmet skal det dog nævnes at du næppe kan have undgået at se tidsforskellen mellem BASIC- og maskinkodeprogrammet. De udfører det samme med ikke helt på samme tid. Indledningen til bogen skulle gerne have givet dig årsagen til dette.

Inden vi går over til at se på dobbeltbyte forøgelsesinstruktioner skal nævnes at " INC " for enkeltbyte påvirker:

ZERO : -som bliver "set", hvis forøgelsen resulterer i tallet nul. Ellers gøres dette flag "reset".

SUBTRAKTION : -bliver "reset", d.v.s. sat til 0.

SIGN : -bliver "set", når bit nr. 7 i forbindelse med forøgelsen bliver "set". Ellers bliver flaget "reset".

Øvrige flageffekter kan ses i Appendix.

B: " INC " for dobbeltbyte:

Disse instruktioner bruges på samme måde som med enkelt byte " INC " blot på dobbelt byte. Eneste forskel er at når de to bytes tælles op med een tages der hensyn til CARRY-flaget. Dette betyder at når der tælles op fra 255 til 256 bliver 'lowbyte' registret "reset" og 'highbyte' registret bliver sat til een. Hensynet til CARRY mærker vi således ikke jf. nedenfor.

Instruktionerne i denne gruppe påvirker IKKE nogen af flagene i " F "-registret.

Der findes instruktioner svarende til samtlige registerpar ialt 6:

Assembler	Decimal kode
INC BC	3
INC DE	19
INC HL	34
INC SP	51
INC IX	221 34
INC IY	253 34

Figur . Oversigt over " INC "-instruktionen for registerpar.

Bemærk at det for indeksregisterne er BASISADRESSEN, som ændres. Dette betyder, at man med instruktionen

LD A.(IX+n)

kan udpege hele tabeller i lageret. Dette sker ved at "n", d.v.s. forskydelsen, er konstant, men at vi ved at forøge " IX "-registret adresserer lageret i kronologisk rækkefølge.

SUBTRAKTIONS - INSTRUKTIONERNE:

Denne gruppe omfatter, meget lig additions instruktionerne, 3 instruktions typer, som er

1. " SUB "-instruktionen,
2. " SBC "-instruktionen og
3. " DEC "-instruktionen,

hvor kun " DEC " og " SBC " kan anvendes på både enkelt og dobbeltbyte.

1. " SUB "-instruktionen:

SUB betyder subtraher eller -træk fra-. Med instruktionen

SUB n

kan du trække et tal fra indholdet i " A "-registret. Du udskifter 'n' med det tal du skal trække fra, men husk at 'n' nødvendigvis må være mellem 0 og 255 (-127 og 128).

Med

SUB r

kan du fratrække indholdet i et register fra indholdet i " A "-registret. Her udskifter du 'r' med det register, hvis indhold du vil fratrække. 'r' kan således være A,B,C,D,E,F,H eller L. Ønsker vi f.eks. at trække 8 fra indholdet i " A "-registret kunne dette ske med:

```
LD B.8
SUB B
```

Dette er imidlertid et meget dårligt eksempel, idet vi lige så godt kunne anvende den direkte fratrækning med " SUB n ". Men i det tilfælde, hvor det tal der skal fratrækkes varierer kan " SUB r " bedst anvendes.

Den sidste variation er instruktionen

SUB (r)

hvor 'r' igen står for indholdet, men i et registerpar. Læsningen af instruktionen kan bedst forståes med et eksempel som det vist nedenfor:

SUB (HL)

Her indeholder " HL "-registret et tal mellem 0 og 65535. Dette tal skal i forbindelse med instruktionen opfattes som en adresse, nemlig adressen på en lagercelle. Parantesen omkring registernavnet betyder at det er indholdet i den adresse som registeret peger på, der skal bruges. Det er således IKKE indholdet i " HL ", der skal bruges. Computeren henter altså indholdet i " HL ", tænker på dette tal, som en adresse i sit lager og henter indholdet i lageradressen. Tallet der står i lageradressen fratrækkes så endelig indholdet i " A "-registret.

Samtlige muligheder af instruktioner er vist nedenfor:

Assembler	Decimal kode
SUB B	144
SUB C	145
SUB D	146
SUB E	147
SUB H	148
SUB L	149
SUB (HL)	150
SUB A	151
SUB (IX+n)	221 150 n
SUB (IY+n)	253 150 n
SUB n	214 n

Tabel 1. Oversigt over " SUB "-instruktionerne.

Du skal ikke lade dig forvirre af at der kun angives eet register i forbindelse med instruktionen. Dette hænger jo sammen med at man kun kan fratrække fra " A "-registret.

Der findes ingen instruktioner til fratrækning af dobbeltbyte tal. Dette skal vi imidlertid lave et lille program, der kan udføre sidst i dette kapitel.

Når vi bruger " SUB "-instruktionen påvirkes " F "-registrets flag som normalt:

SIGN : -flaget påvirkes af resultatets fortegn efter de almindelige regler vedrørende bit nr. 7.

ZERO : -bliver "set", når resultatet bliver nul, ellers "reset".

CARRY : -bliver "reset", hvis det tal der står i " A "-registret er mindre end eller lig med det tal, som skal fratrækkes. CARRY bliver så "set" i det modsatte tilfælde.

SUBTRAKTION : -bliver "set".

Se Appendix vedrørende de øvrige flag.

Nedenfor er vist et lille eksempel, som skulle kunne vise dig effekten på CARRY-flaget:

```
Assembler        Decimal kode

LD A.230         62 230
LD H.240         38 240
SUB A.H          148
ADC 0            206 0
LD (40000).A     50 64 156
RET              201
```

Figur . Illustration af CARRY-flageffekten.

Vi trækker altså 240 fra 230, hvilket giver -10. Idet vi adderer med CARRY (" ADC ") i den efterfølgende instruktion får vi adderet een til, HVIS CARRY-flaget er "set".

Når du har indtastet og fået udført nedenstående BASIC-program skulle du gerne se tallet 247 på skærmen ????.

```
Linienr. Kommando

    10 REM************************************
    15 REM****** ILLUSTRATION AF SUB ********
    19 REM************************************
    20 FOR a=1 TO 11
    30 READ f
    40 POKE a+42999,f
    50 NEXT a
    60 DATA 62,230,38,240,148,206,0,50,
       64,156,201
    70 CALL 43000
    80 PRINT PEEK (40000)
    90 END
    95 REM************************************
```

Når vi fratrækker et tal fra " A "-registret, der er større end det der står i " A "-registret bliver CARRY-flaget "set". Dette kan vi se via flageffekten ovenfor. Da vi adderer med " ADC " lægger vi således een til, hvilket jo er CARRY-flaget. Tager vi denne fra skulle det rigtige resultat være 247-1=246.

Forklaringen på de 246 findes i den måde hvorpå computeren lagrer negative tal. Maskinen lagrer ikke et minus, hvilket ville kræve en hel lagercelle. I stedet lagrer den et signal, hvilket egentlig blot er en bit - nemlig bit nr. 7. Når denne er "set" har computeren lagret et negativt tal. - Ingen problemer ? - hvad nu når vi lagrer f.eks. tallet 130. Det ville se således ud i lagercellen:

```
bit nr.: 7 6 5 4 3 2 1 0

         1 0 0 0 0 0 1 0
```

Efter hvad vi netop har sagt skulle dette tal være et negativt tal ?! - noget er galt -. Det der er galt er at vi må indføre et system, der angiver hvornår vi arbejder med negative tal og hvornår vi ikke gør det. Dette er noget DU selv må holde rede på. I hele denne bog benytter vi kun ganske få gange negative tal så i de fleste tilfælde vil det ikke volde problemer. Men skal du til at bruge dem må du være påpasselig.

Uanset om du anvender negative tal eller ej - så gør computeren det. Dette er grunden til at den udskriver de 246. Lad os se på dette tal:

```
bit nr.:  7 6 5 4 3 2 1 0

          1 1 1 1 0 1 1 0
```

Her kan vi se at bit nr. 7 er "set" og det må således kunne være et negativt tal. Når vi har konstateret at det må være et negativt tal kan vi lave det om til et decimalt tal ved at trække 256 fra tallet. Herved får vi 246-256= -10.

Hermed skulle du have forstået anvendelsen af såvel CARRY-flaget som negative tal.

B: " SBC " - INSTRUKTIONEN:

Som nævnt i indledningen kan instruktionen anvendes til såvel enkelt- som dobbeltbyte subtraktion.

a) " SBC " for enkeltbyte:

SBC svarer til ADC, idet SBC tager hensyn til CARRY-flaget i fratrækningen. Dette betyder, som vi skal se, at vi kan foretage een subtraktion og tage hensyn til om en tidligere subtraktion foretog et lån i en højere byte.

Der findes 11 instruktioner, som vist nedenfor:

Assembler	Decimal kode
SBC B	152
SBC C	153
SBC D	154
SBC E	155
SBC H	156
SBC L	157
SBC (HL)	158
SBC A	159
SBC (IX+n)	221 158 n
SBC (IY+n)	253 158 n
SBC n	222 n

Tabel 1. Oversigt over " SBC "-instruktionerne.

Idet " SBC " kun afviger ganske lidt fra " SUB " vil vi kun se på denne forskel. Nedenfor ses to subtraktioner den ene med " SUB " - den anden med " SUB ":

eks. 1:

Assembler	Decimal kode
LD A.20	62 20
LD H.210	38 210
SUB H	148
SUB H	148
LD (40000).A	50 64 156
RET	201

eks. 2:

Assembler	Decimal kode
LD A.20	62 20
LD H.210	38 210
SUB H	148
SBC H	156
LD (40001).A	50 65 156
RET	201

BASIC-loaderen er vist nedenfor:

Linienr. Kommando

```
10 REM************************************
20 REM*** FORSKEL MELLEM SUB OG SBC ******
30 REM************************************
40 FOR a=1 TO 18
50 READ f
60 POKE a+42999,f
70 NEXT a
75 DATA 62,20,38,210,148,148,50,64,156,201
76 DATA 62,20,38,210,148,156,50,65,156,201
80 CALL 43000
82 PRINT PEEK (40000)
84 CALL 43010
86 PRINT PEEK (40001)
90 REM************************************
```

Når du har fået programmet udført skulle du gerne se tallene 111 og 112 stå under hinanden på skærmen. Programmets funktion er alene at trække et tal fra indholdet i " A "-registret to gange.

Først trækkes de 210 (i " H " -registret) fra de 20 (I " A "-registret). Hermed skulle " A "-registret indeholde -190 eller som vi nu ved 256-190= 66. Dette på grund af det vi talte om i det forrige afsnit.

Så trækkes de 210 fra igen og det er her at der er forskel på de to programmer. I eksempel 1 tages der ikke hensyn til CARRY-flaget, d.v.s. hvorvidt den forrige subtraktion skulle have gjort det "set" eller ej. Dette er så også årsagen til forskellen på de to resultater vist på skærmen. Tallet i " A "-registret, d.v.s. de 66, fratrækkes de 210 hvilket giver -144 eller 112 (256-144=112). Men da vi i forrige subtraktion 'lånte' af en højere byte fratrækkes de 112 een i det andet eksempel.

b) " SBC " for registerpar:

Her findes 4 instruktioner til fratrækning af 16-bit tal. Subtraktionen kan imidlertid kun ske FRA " HL "-registerparret. Dette betyder at der vil være behov for at flytte de to tal der skal trækkes fra hinanden til de implicerede registerpar.

Instruktionerne er

Assembler	Decimal kode
SBC HL.BC	237 68
SBC HL.DE	237 85
SBC HL.DE	237 102
SBC HL.SP	237 119

Figur . " SBC "-instruktionerne for registerpar.

Med disse instruktioner er der mulighed for at foretage subtraktion af 16-bit tal. Nedenfor er vist hvordan vi meget let kan foretage dette:

Assembler	Decimal kode
LD HL.(adresse 11)	42 n n
LD DE.(adresse 21)	237 91 n n
AND A	167
SBC HL.DE	237 82
LD (adresse 31).HL	34 n n

hvor du nedenfor kan se hvordan tallene, der skal subtraheres skal placeres i lageret. Programmet skulle være let at forstå; måske skal det siges at " AND A "-instruktionen er der for at gøre CARRY-flaget "reset", idet dette flag jo påvirker subtraktionen.

Når du skal bruge programmet skal du placere de to tal som vist nedenfor:

adresse 11, adresse 21 og adresse 31

indeholder lowbytene.

adresse 1h, adresse 2h og adresse 3h

indeholder highbytene.

Adresse 1 og 2 indeholder lowbytene af tallene der skal adderes, mens resultatet står i adresse 3:

```
lageradresser:      ....
                    ....
                    adresse 1h
                    adresse 1l
                    ....

                    ....
                    adresse 2h
                    adresse 2l
                    ....

                    ....
                    adresse 3h
                    adresse 3l
                    .....
```

På denne måde kan vi placere resultatet i den lagercelle, der bedst passer os.

FLAGEFFEKT:

SUBTRAKTION: -dette flag gøres naturligt nok "set".

SIGN : -hvis resultatet af subtraktionen gør bit nr. 7 "set" gøres SIGN-flaget "set".

ZERO : -bliver resultatet nul gøres ZERO-flaget "set" ellers "reset".

CARRY : -bliver "set", hvis tallet der fratrækkes er størst ellers "reset".

" DEC " - INSTRUKTIONEN:

Dec ordet er forkortelsen for decrement eller 'formindsk' (med een). Instruktionen kan, som " SBC "-instruktionen bruges på såvel enkelt- som dobbeltbyte tal.

a) " DEC " for enkelt bytes:

Med

DEC r

kan du nedtælle indholdet i et register. Her udskifter du 'r' med det register, hvis indhold du vil nedtælle 'r' kan således være A,B,C,D,E,F,H eller L. Ønsker vi f.eks. at benytte " B "-registret som 'tæller' kunne dette ske med:

```
LD B.8
DEC B
```

Den anden variation er instruktionen

DEC (r)

hvor 'r' igen står for indholdet, men i et registerpar. Læsningen af instruktionen kan bedst forståes med et eksempel som det vist nedenfor:

DEC (HL)

Her indeholder " HL "-registret et tal mellem 0 og 65535. Dette tal skal i forbindelse med instruktionen opfattes som en adresse, nemlig adressen på en lagercelle. Parantesen omkring registernavnet betyder at det er indholdet i den adresse som registeret peger på, der skal bruges. Det er således IKKE indholdet i " HL ", der skal bruges. Computeren henter altså indholdet i " HL ", tænker på dette tal, som en adresse i sit lager og henter indholdet i lageradressen. Tallet der står i lageradressen nedtælles så endelig indholdet i " A "-registret.

Samtlige muligheder af instruktioner er vist nedenfor:

Assembler	Decimal kode
DEC B	5
DEC C	13
DEC D	21
DEC E	29
DEC H	34
DEC L	45
DEC (HL)	51
DEC A	61
DEC (IX+n)	221 51 n
DEC (IY+n)	253 51 n

Tabel 1. Oversigt over " DEC "-instruktionerne.

Nedenfor er vist hvordan instruktionen påvirker " F "-registrets flag:

SUBTRAKTION: -dette flag gøres "set", idet denne instruktionen jo mindsker indholdet i det angivne register (eller lagercelle).

SIGN : -dette flag vil blive "set", hvis resultatet er negativt, d.v.s. bit nr. 7 er "set".

ZERO : -som normalt bliver dette flag "set" ved resultatet nul.

a) " DEC " for dobbelt-bytes:

Med

DEC registerpar

kan du nedtælle indholdet i et registerpar med een. Her udskifter du 'r' med det register, hvis indhold du vil nedtælle 'r' kan således være BC, DE, HL, IX, IY og SP. Ønsker vi f.eks. at benytte " BC "-registret som 'tæller' kunne dette ske med:

```
LD BC.80
DEC BC
```

Instruktionerne er vist nedenfor:

Assembler	Decimal kode
DEC BC	11
DEC DE	27
DEC HL	43
DEC IX	221 43
DEC IY	253 43
DEC SP	59

Vi kan illustrere anvendelse af " DEC "-instruktionen ved at 'cleare' skærmbilledet via at lagre en bestemt værdi, f.eks. nul i samtlige lagerceller, som hører til skærmbilledet:

```
          Assembler      Decimal kode

          LD BC.16384    1 0 64
          LD HL.49152    33 0 192
          XOR A          175
loop:     LD (HL).A      119
          DEC BC         11
          JR NZ,loop     32 252
          RET            201
```

Figur . Sletning af skærmbilledet.

Afhængig af hvilket "MODE" du anvender, d.v.s.

MODE 0 - 20 bogstaver pr. linie i 16 farver.
MODE 1 - 40 bogstaver pr. linie i 4 farver.
MODE 2 - 80 bogstaver pr. linie i 2 farver.

stiger antallet af bytes, som skærmbilledet skal anvende. Jo flere bogstaver pr. linie, jo mere kræver bogstaverne af lageret - heroverfor står at jo flere farver du anvender, des flere bytes bruges. Anvender du de ovennævnte værdier får du slettet skærmen uafhængigt af hvilket " MODE " du er i.

Dobbelt byte instruktionerne påvirker ikke flagene.

SAMMENLIGNINGS INSTRUKTIONERNE:

Sammenligning hedder "Compare" på engelsk, hvilket netop er navnet på denne gruppe instruktioner.

Denne instruktionsgruppe giver mulighed for at sammenligne en konstant angivet direkte eller indirekte med indholdet i " A "-registret. Konstanten kan enten være angivet direkte med et tal eller indirekte via et register eller en lageradresse.

De 11 instruktioner der kan bruges til at sammenligne tal er vist nedenfor:

Assembler	Decimal kode
CP B	184
CP C	185
CP D	186
CP E	187
CP H	188
CP L	189
CP (HL)	190
CP A	191
CP (IX+n)	221 190 n
CP (IY+n)	253 190 n

Figur . Sammenlignings instruktionerne for enkelt byte.

Som du kan se nævnes kun det ene tal, der skal benyttes i sammenligningen - det andet står i " A "-registret. Hvorvidt " A "-registret indeholder tallet, der testes imod er dit valg.

Når instruktionen udføres sker det næsten som " SUB "-instruktionen, som blev gennemgået tidligere. Der er naturligvis forskelle. Forskellen består i at indholdet i " A "-registret IKKE ændres i forbindelse med instruktionen. Ved " SUB "-instruktionen indeholdet " A "-registret jo resultatet af subtraktionen.

Svarende til " SUB " påvirkes flagene i " F "-registret. Det tal du angiver i instruktionen, enten direkte eller indirekte, bliver trukket fra indholdet i " A "-registret. Flagene vil så ændres svarende til resultatet, eksempelvis ved resultatet nul "set"tes ZERO-flaget. Som ved subtraktion gøres SUBTRAKTIONS flaget "set" til trods for at sammenligning jo ikke er en reel fratrækning.

En af anvendelserne for sammenligningsinstruktionerne er muligheden for at søge efter en bestemt byte eller flere i lageret. Behovet for dette har (og er) meget stort, hvorfor man har udviklet 4 specielle `halv- og helautomatiske instruktioner til at klare disse opgaver. Disse kaldes "bloksøgningsinstruktionerne" og vi skal se på dem i slutningen af dette afsnit.

Vi kan imilertid alene med " CP "-instruktionen lave vort eget søgeprogram. Ved at pege på en startadresse, teste indholdet i denne med den byte vi ønsker at finde, øge adressenummeret og prøve igen hvis vi ikke fandt hvad vi søgte. Testen sker med " CP " mens øgningen af adressenummeret sker med " INC ". Efterhånden som vi kommer gennem instruktionssættet vil du opdage at der ofte er flere muligheder for løsning af problemerne du kommer ud for. Det er der også her, eksempelvis kunne man bruge en additionsinstruktion (" ADD "," ADC ").

Vi kan som et eksempel søge i computerens ROM, hvor operativsystemet ligger. Vi starter søgningen i adresse 0 og fortsætter indtil vi har fundet første forekomst af byten 255. Som adresse peger bruger vi " HL "-registret.

Assembler	Decimal kode
LD BC.32649	1 137 127
OUT (C).C	237 73
LD HL.0	33 0 0
LD A.(HL)	126
CP 255	254 255
LD (40000).HL	34 56 164
RET Z	200
INC HL	35
JR LOOP	24 247

Figur .Enkelt byte søgeprogram.

Der er flere ting i dette lille program, som du bør lægge mærke til. Først den direkte angivelse af det tal vi søger efter. Dette program er ikke så fleksibelt at det kan søge efter andet end tallet 255. Brugte vi i stedet et registers indhold kunne vi jo ændre indholdet i registret. Her skal vi `ramme` netop den adresse som " CP " instruktionen står i for at ændre tallet. Foruden dette besvær er metoder ABSOLUT IKKE anbefalelsesværdig. Det gør programmet mere ulæseligt.

Bemærk også at den adresse der aflæses, d.v.s. indholdet i " HL "-registret, hele tiden lagres i adresse 40000/40001. Når den søgte byte er fundet kan vi aflæse indholdet i de ovennævnte adresse og få oplysning om adressen. En anden mulighed var at udskrive adressen på skærmen, hvilket vi skal se på senere.

Endelige skal du bemærke flageffekten, som jo er nødvendig for såvel programmets loop som dets afslutning. Hvis " LD "-instruktionen havde påvirket flagene i " F "-registret måtte vi omprogrammere det hele. Vigtigst er det dog at IKKE zeroflaget påvirkes, idet det jo er dette som vi bruger og ikke de andre.

Aflæsningen af maskinens operativsystem er tilfældigt valgt og kunne derfor lige så godt være en tekst indtastet i et tekstbehandlingsprogram. Således kunne tallet vi søger efter være koden for et eller flere bogstaver.

Ønsker du at søge efter en bestemt "sekvens", d.v.s. rækkefølge af tal (eller koder) kunne dette ske som nedenfor vist:

```
                Assembler

                LD BC.32649
                OUT (C).C
                LD BC.`GRÆNSE`
                LD HL.0
LOOP1:          CP `1.TAL`
                JR Z.LOOP
                INC HL
                LD A.H
                CP B
                JR NZ.LOOP1
                LD A.L
                CP C
                LD (40000).HL
                RET Z
                JR LOOP1
LOOP:           INC HL
                CP `2.TAL`
                JR Z.LOOP2.....
```

Figur . Program til søgning af sekvens af koder.

I figuren nedenfor er vist BASIC programmet til at lagre maskinkoden i maskinenslager.

BLOKSØGNINGS INSTRUKTIONER:

Der er 4 bloksøgningsinstruktioner, som lettest beskrives som sammensat af flere instruktioner. De to første instruktioner er "halvautomatiske":

Assembler	Decimal kode
CPI	237 161
CPD	237 169

Tabel. Oversigt over bloksammenlignings-instruktionerne.

som er sammensat af " CP (HL) "," INC HL "/" DEC HL " og " DEC BC ".

" CPI " er forkortet for ComPare (sammenlign) indholdet i med " A "-registret og Increment (øg) HL-registret, d.v.s. vi behøver blot at lave et loop samt testen af flagene:

```
CPI:    CP (HL)
        INC HL
        DEC BC
```

Svarende til dette er " CPD " forkortet for ComPare, men med Decrement (øg) af " HL "-registret. Du kan således adskille de to instruktioner ved det sidste bogstav i navnet:

```
CPD:    CP (HL)
        DEC HL
        DEC BC
```

Bemærk at " BC "-registret i begge tilfælde er tæller, d.v.s. det tal du har lagret i dette register vil blive nedtalt for hver sammenligning. Det vil derfor være naturligt at du tester ZERO-flaget for at få mulighed for at angive en grænse for den " største " adresse du vil gennemsøge.

Den måde flagene i " F "-registret påvirkes er en smule kompliceret. Da både " CP " og " DEC "-instruktionerne påvirker ZERO-flagene ville det ikke være muligt at afgøre om f.eks. ZERO-flaget var blevet "set" som følge af den ene eller den anden instruktion. Derfor er det PARITETS/OVERFLOW flaget du skal teste for at finde ud af om BC-registret er nul. Når dette flag er "reset" har " BC "-registret nået nul. Således kan ZERO-flaget koncentreres om " CP "-instruktionen. ZERO-flaget vil blive "set", hvis indholdet i " A "-registret er lig med indholdet i den lagercelle som " HL "-registret peger på.

Herudover bliver SUBTRAKTIONS-flaget "set", mens FORTEGNS- og HALVCARRY-flagene bliver påvirket på normal måde. CARRY ændres ikke.

Den anden halvautomatiske bloksøgningsinstruktion påvirker flagene på samme måde som " CPI ". Eneste forskel er således kun at " HL "-registret mindskes med een for hver gang instruktionen udføres.

De to automatiske instruktioner er:

Assembler	Decimal kode
CPIR	237 177
CPDR	237 185

og kaldes således ("automatiske") på grund af at instruktionerne i sig selv udgør et loop.

" CPIR "-instruktionen er sammensat af

```
LOOP:   CP (HL)
        INC HL
        DJNZ LOOP
```

hvilket betyder at du inden instruktionen skal forberede denne, hvilket vil sige:

1) Lagre startadressen i " HL "-registret
2) Lagre antallet af bytes der skal gennemsøges i " BC "-registret.
3) Lagre den byte (kode) du ønsker at søge efter i " A "-registret.

og så udføre " CPIR "-instruktionen.

Igen er der `problemer` med ZERO-flaget. Her vil instruktionen først `standse`, d.v.s. computeren hopper til næste instruktion, når ENTEN indholdet i " A "-registret er lig med indholdet i den lagercelle der peges på via " HL "-registret, ELLER " BC "-registret er nedtalt til nul.

Dette betyder at du ikke umiddelbart ved hvilken af de to muligheder der har gjort ZERO-flaget "set". Dette kan løses ved at teste PARITETS/OVERFLOW-flaget. Er dette flag "reset" betyder det at " BC "-registret er nul.

En anden mulighed er naturligvis også at finde ud af om " BC "-registret rent faktisk er nul, etc.

" CPDR "-instruktionen svarer til den ovenfor nævnte blot med den forskel at " HL "-registret `nedtælles`. Dette betyder at du angiver `toppen` af det område du ønsker gennemsøgt. Fra denne adresse arbejder maskinen sig ned indtil instruktionen standses via fundet af den søgte byte eller at " BC " nedtælles til nul.

Påvirkningen af flagene svarer til det omtalte for " CPIR "-instruktionen.

LOGISKE INSTRUKTIONER

Disse instruktioner, som omfatter

1. " AND " -instruktionen,
2. " OR " -instruktionen og
3. " XOR " -instruktionen

der bruges til, på forskellig måde, udfra de to bytes som sammensættes og de regler der gælder for instruktionen, at styre enkelte eller flere bit i resultatbyten. Denne byte er altid " A "-registret. Instruktionerne sammenligner bitvis og uafhængigt af resultatet af de øvrige sammenligninger.

" A "-registret er altid den ene part i instruktionen, og samtidig resultatregistret. Byten i " A "-registret kan så sammenlignes med bytes angivet direkte eller indirekte via enten et register,et tal eller en lageradresse.

" AND " - INSTRUKTIONERNE

Navnet AND kan oversættes til - og -. Når to bytes skal sammenlignes med " AND "-instruktionen bliver resultatet udregnet bitvis, d.v.s. begge bytes' bit nr. 0 sammenlignes, derefter begge bytes' bit nr. 1, osv.

Resultatet bliver, når

- begge bit er 'set' (=0), en 'set' bit som resultat i " A " registret på netop den plads, hvor det sammenlignede bitpar står.
- en eller begge bit er 'reset', en 'reset' (=0) bit.

Ser vi på de fire muligheder der således findes bliver resultaterne

```
          1 1 0 0
AND       1 0 1 0
          1 0 0 0
```

Et eksempel kan vise dette i praksis. Vi sammenligner tallene

1) 0000 1101 2) 1100 0101

med " AND ". Resultatet bliver

```
     1)   0 0 0 0   1 1 0 1
AND  2)   1 1 0 0   0 1 0 1
          0 0 0 0   0 1 0 1
```

Når vi sammenligner resultatet kan vi begynde for- eller bagfra; det har ingen betydning, idet resultatet beregnes for de enkelte bit uafhængigt af de andre. Sammensætningen (lodret) 0-0, 0-1 og 1-0 giver 0, d.v.s. en 'reset' bit. Sammensætningen 1-1 giver 1, d.v.s. en 'set' bit.

Vi kan altså se at kun, når BEGGE bit er 'set' vil resultatet af en AND-instruktion blive en 'set' bit. Dette må betyde at hvis vi laver en " AND "-sammenligning af en byte med sig selv vil resultatet blive nøjagtigt som byten selv.

```
     1)  0101 0111
AND  2)  0101 0111
         0101 0111
```

Nedenfor er vist et program, der kan vise at AMSTRAD er enig med det vi netop har beregnet.

```
DI
LD BC.32649
OUT (C).C
LD A.(43050)
LD HL.43050
AND (HL)
CALL 5120
EI
RET
```

Figur 1. Illustration af " AND "-instruktionen.

Som sædvanligt er linie 1-3 og 8 kun af betydning for vores anvendelse af AMSTRADs operativsystems udskrivnings-rutine.

Nedenfor er vist BASIC-programmet, der skal lagre maskinkode instruktionerne i lagret. I programmet skal du indtaste et tal, som er det tal der skal foretages " AND "-sammenligning med. Dette tal lagres i lagercelle 43050, som iøvrigt er tilfældigt valgt, d.v.s. du kan vælge en hvilken som helst anden. Maskinkodeprogrammet aflæser imidlertid denne adresse og lagrer en kopi af tallet i " A "-registret. Herefter sættes " HL "-registret også til at pege på lagercelle 43050. Endelig udføres AND-instruktonen, her i den indirekte version, d.v.s. hvor vi ikke direkte angiver tallet som " A "-registret. I stedet peger " HL "-registret på den adressecelle, der skal bruges. På denne måde får vi sammenlignet indholdet i celle 43050 med sig selv.

Hvad mener du nu resultatet skulle blive ?

Den værdi du indtastede selvfølgelig ! - stemte det med det du så på AMSTRAD-monitoren, ...nå ikke ?... Hvorfor mon ikke ?

Det skyldes at AMSTRAD har defineret ethvert tal mellem 0 og 255 til et eller andet symbol. Disse symboler samt de tilhørende talværdier kan ses i brugermanualens Appendix.

AND betyder logisk addition. Med instruktionen

AND n

kan du logisk sammenligne to tal; det ene indholdet i " A " -registret. Du udskifter 'n' med det tal du skal trække fra, men husk at 'n' nødvendigvis må være mellem 0 og 255 (-127 og 128).

Med

AND r

kan du sammenligne indholdet i et register med indholdet i " A "-registret. Her udskifter du 'r' med det register, hvis indhold du vil sammenligne. 'r' kan således være A,B,C,D,E,F,H eller L. Ønsker vi f.eks. at sammenligne 8 med indholdet i " A "-registret med " AND " kunne dette ske med:

```
LD B.8
AND B
```

Dette er imidlertid et meget dårligt eksempel, idet vi lige så godt kunne anvende den direkte sammenligning med "AND n". Men i det tilfælde, hvor det tal der skal sammenlignes med varierer kan " AND r " bedst anvendes.

Den sidste variation er instruktionen

AND (r)

hvor 'r' igen står for indholdet, men i et registerpar. Læsningen af instruktionen kan bedst forståes med et eksempel som det vist nedenfor:

AND (HL)

Her indeholder " HL "-registret et tal mellem 0 og 65535. Dette tal skal i forbindelse med instruktionen opfattes som en adresse, nemlig adressen på en lagercelle. Parantesen omkring registernavnet betyder at det er indholdet i den adresse som registeret peger på, der skal bruges. Det er således IKKE indholdet i " HL ", der skal bruges. Computeren henter altså indholdet i " HL ", tænker på dette tal, som en adresse i sit lager og henter indholdet i lageradressen. Tallet der står i lageradressen sammenlignes så endelig indholdet i " A"-registret.

Der findes ialt 11 metoder at angive den byte, som skal sammenlignes med indholdet i " A "-registret, når dette skal gøres med " AND "-instruktionen. Disse 11 metoder er vist nedenfor:

Assembler	Decimal kode
AND B	160
AND C	161
AND D	162
AND E	163
AND H	164
AND L	165
AND (HL)	166
AND A	167
AND (IX+n)	221 166 n
AND (IY+n)	253 166 n
AND n	230 n

Tabel 1. Oversigt over " AND "-instruktionerne.

Efter AND-ordet angives, hvilke tal der skal " AND "-sammenlignes. Bemærk at du kun har mulighed for at pege på eet tal i forbindelse med instruktionen. Dette hænger sammen med at det tal du angiver ALTID bliver " AND "-sammenlignet med indholdet i " A "-registret. Indholdet i det register, f.eks. H, eller indholdet i den lageradresse, f.eks. ved (HL), ændres IKKE i forbindelse med " AND "-instruktionen. Det er KUN indholdet i " A "-registret samt flagene i " F "-registret, som påvirkes af denne logiske instruktion.

FLAGEFFEKT:

CARRY : -flaget bliver "reset".
ZERO : -flaget fungerer som normalt, d.v.s. det bliver "set", når instruktionen gør samtlige bit "reset"; ellers gøres flaget "reset".
SIGN : -flaget vil blive "set", hvis bit nr. 7 er "set" som følge af "AND"-instruktionen. Dette er jo kun muligt, hvis begge bit (nr. 7) er "set" inden "AND"-instruktionen.
PARITET -flaget vil blive "set", hvis der er lige paritet, d.v.s. et lige antal "set" bits i resultatbyten. Ellers gøres flaget "reset".

De øvrige flags påvirkning kan ses i Appendix.

Du skulle nu kunne se nytten af de logiske instruktioner, specielt hvad angår flagene. Som før nævnt vil f.eks. instruktonen " AND A ", d.v.s. " A "-registret sammenlignet med sig selv , betyde at indholdet i " A "-registret forbliver uændret, men at CARRY-flaget bliver "reset". Arsagen til at " A " registret forbliver uændret skulle du kunne se ud af regnereglerne vist på forrige side. Hver "set" bit i " A "-registret sammenlignes med en "set" bit, hvilket igen giver en "set" bit. Tilsvarende med de "reset"te bit i " A "-registret. Disse sammenlignes med "reset"te bits, som igen giver "reset"te bits.

Instruktionen " AND 0 ", d.v.s. " A "-registret AND-sammenlignet med nul skulle:

1. gøre hele " A "-registret "reset", d.v.s. nul.
2. gøre ZERO-flaget "set".

Dette kan du få at se ved at prøve nedenstående program.

```
Assembler          Decimal kode

LD BC.32649        1 137 127
OUT (C).C          237 73
LD A.100           62 100
AND 0              230 0
ADD 100            198 100
...
```

Figur .Illustration af " AND 0 ".

I stedet for instruktionen " AND " med tallet nul kunne du angive netop det tal der peger på den eller de bit du ønsker at påvirke. Eksempelvis vil

```
.
.
AND 128
JP P,adresse
```

teste om " A "-registrets bit nr. 7 er "set" eller "reset". Således kan du styre resultatet i " A "-registret med de logiske instruktioner.

Hvis du eksempelvis ønsker at et tal ikke må overstige en grænse, eksempelvis 15, kan du "AND"-sammenligne tallet der skal testes med " AND 15 ". Binært ser tallet 15 således ud:

0 0 0 0 1 1 1 1 (=15)

hvilket betyder, j.f. reglerne for " AND ", at kun de første 4 bits, d.v.s. bit nr. 0 til 3 forbliver som de hele tiden har været. De øvrige bits vil, ligegyldigt hvad de var forud, blive "reset". Hermed kan tallet IKKE blive større end 15.

" OR " - INSTRUKTIONEN:

Ordet OR betyder -eller-. Når to tal sammenlignes med " OR "-instruktionen bliver resultatet:

1. en "set" bit -når ENTEN den ene ELLER begge bit inden " OR " er "set".
2. en "reset" bit -når BEGGE bit er "reset" inden operationen.

De fire muligheder ser således ud:

```
      0 0 1 1
OR    0 1 0 1
      0 1 1 1
```

Vi får således KUN en "reset" bit, hvis begge bit før sammenligningen er "set".

De 11 metoder, der også findes i denne gruppe, kan anvendes svarende til "OR"-instruktionen, d.v.s. at vi kan angive den ene part af de to tal til sammenligningen. Den anden er indholdet i " A "-registret.

Assembler	Decimal kode
OR B	160
OR C	161
OR D	162
OR E	163
OR H	164
OR L	165
OR (HL)	166
OR A	167
OR (IX+n)	221 166 n
OR (IY+n)	253 166 n
OR n	230 n

Tabel 1. Oversigt over " OR "-instruktionerne.

Efter OR-ordet angives, hvilke tal der skal " OR "-sammenlignes. Bemærk at du kun har mulighed for at pege på eet tal i forbindelse med instruktionen. Dette hænger sammen med at det tal du angiver ALTID bliver " OR "-sammenlignet med indholdet i " A "-registret. Indholdet i det register, f.eks. H, eller indholdet i den lageradresse, f.eks. ved (HL), ændres IKKE i forbindelse med " OR "-instruktionen. Det er KUN indholdet i

" A "-registret samt flagene i " F "-registret, som påvirkes af denne logiske instruktion.

FLAGEFFEKT:

CARRY : -flaget bliver "reset".
ZERO : -flaget fungerer som normalt, d.v.s. det bliver "set", når instruktionen gør samtlige bit "reset"; ellers gøres flaget "reset".
SIGN : -flaget vil blive "set", hvis bit nr. 7 er "set" som følge af "OR"-instruktionen. Dette er jo kun muligt, hvis begge bit (nr. 7) er "set" inden "OR"-instruktionen.
PARITET -flaget vil blive "set", hvis der er lige paritet, d.v.s. et lige antal "set" bits i resultatbyten. Ellers gøres flaget "reset".

De øvrige flags påvirkning kan ses i Appendix.

Som med "AND"-instruktionen kan også "OR" benyttes til at styre bits i A-registret samt flagene i " F "-registret.

Instruktionen " OR A " vil ikke påvirke indholdet i " A "-registret, men vil afhængig af indholdet i " A "-registret påvirke ZERO- eller SIGN-flaget. Dette svarer til instruktionen "OR 0". Ønskes indholdet i " A "-registret gjort maksimalt vil instruktionen "OR 255" gøre dette.

" XOR " INSTRUKTIONEN:

Dette er en speciel udgave af "OR"-instruktonen, hvilket også fremgår af instruktionen navn, som er 'exklusive OR'. Den adskiller sig fra den almindelige "OR"-instruktion ved ikke at gøre resultatbitten "set" ved sammenligning af to "set" bit.

De fire muligheder er vist nedenfor:

```
      0 0 1 1
XOR   0 1 0 1
      0 1 1 0
```

d.v.s. resultatet

1. en "set" bit -når den ene bit inden " OR " er "set".

2. en "reset" bit -når ENTEN begge bit er "reset" inden operationen eller når begge er "set" inden operationen.

Nedenfor er samtlige registre, der kan sammenlignes med " XOR ".

Assembler	Decimal kode
XOR B	168
XOR C	169
XOR D	170
XOR E	171
XOR H	172
XOR L	173
XOR (HL)	174
XOR A	175
XOR (IX+n)	221 174 n
XOR (IY+n)	253 174 n
XOR n	238 n

Tabel 1. Oversigt over " XOR "-instruktionerne.

Efter XOR-ordet angives, hvilke tal der skal " XOR "-sammenlignes. Bemærk at du kun har mulighed for at pege på eet tal i forbindelse med instruktionen. Dette hænger sammen med at det tal du angiver ALTID bliver " XOR "-sammenlignet med indholdet i " A "-registret. Indholdet i det register, f.eks. H, eller indholdet i den lageradresse, f.eks. ved (HL), ændres IKKE

i forbindelse med " XOR "-instruktionen. Det er KUN indholdet i " A "-registret samt flagene i " F "-registret, som påvirkes af denne logiske instruktion.

FLAGEFFEKT:

CARRY : -flaget bliver "reset".
ZERO : -flaget fungerer som normalt, d.v.s. det bliver "set", når instruktionen gør samtlige bit "reset"; ellers gøres flaget "reset".
SIGN : -flaget vil blive "set", hvis bit nr. 7 er "set" som følge af "XOR"-instruktionen. Dette er jo kun muligt, hvis begge bit (nr. 7) er "set" inden "XOR"-instruktionen.
PARITET -flaget vil blive "set", hvis der er lige paritet, d.v.s. et lige antal "set" bits i resultatbyten. Ellers gøres flaget "reset".

De øvrige flags påvirkning kan ses i Appendix.

JUMP - INSTRUKTIONERNE:

Indtil nu har de instruktioner vi har set på kun givet os mulighed for at lave programmer der skulle udføres linie for linie, første før nr. to osv. Med JUMP-instruktioner får vi mulighed for at 'springe' instruktioner m.v. over. JUMP betyder -hop- og instruktionen udfører som sagt et hop i rækken af instruktioner der skal udføres. Med denne uundværlige og meget hyppigt anvendte instruktion sker således det samme som med BASIC-kommandoen.

Ligesom man i BASIC kan gøre kommandoen betinget af et eller andet via f.eks. en IF/THEN, IF/THEN/ELSE etc. kan JUMP-instruktionen gøres betinget af flag i " F "-registret. Hoppet kan f.eks. være betinget af at ZERO-flaget er "set".

Der er ialt 17 JUMP-instruktioner, som det kan ses nedenfor. Bemærk at de første 12 skrives med " JP ", som du skal holde adskildt fra " JR "-instruktionen. JP er forkortelsen for

JUMP POINT

eller at hoppe til et adresse (point), der er angivet i instruktionen. Heroverfor står " JR ", som betyder

JUMP RELATIVE

eller hop et antal bytes frem. Forskellen består i at du i den sidste egentlig ikke direkte angiver en adresse, men blot et antal bytes der skal hoppes frem. Denne instruktion bliver herved hurtigere for computeren at udføre, idet den kræver færre bytes.

Instruktionerne i gruppen ses nedenfor:

	Assembler	Decimal kode
ABSOLUTTE HOP:		
direkte hop:	JP adresse	195 n n
indirekte hop:	JP (HL)	233
	JP (IX)	221 233
	JP (IY)	253 233
betinget hop:	JP C.adresse	218 n n
	JP NC.adresse	210 n n
	JP Z.adresse	202 n n
	JP NZ.adresse	194 n n
	JP P.adresse	242 n n
	JP M.adresse	250 n n
	JP PE.adresse	234 n n
	JP PO.adresse	226 n n
RELATIVE HOP:		
direkte hop:	JR forskydning	24 n
betinget hop:	JR C.forskydning	56 n
	JR NC.forskydning	48 n
	JR Z.forskydning	40 n
	JR NZ.forskydning	32 n

Figur . Oversigt over hop-instruktionerne.

Adressen der står nævnt efter hop-instruktionen skal angives på en ganske bestemt måde for at hoppet vil blive udført korrekt. Når du har adressetallet skal dette splittes op i en low- og en high byte. Lowbyten skal så stå FØRST, d.v.s. lige efter koden for instruktionen - herefter high byten. Ønsker du eksempelvis at hoppe til adresse 42000 skal vi først splitte denne adresse op i low/high byte:

Highbyte: INT (42000 / 256) = 164

Low-byte: ((42000 / 256) - highbyte) * 256 = 16

Du kan altid kontrollere din beregning med formelen

Adresse = lowbyte + 256 * highbyte

d.v.s.

42000 = 16 + 256 * 164

Skal vi lave et absolut direkte hop til adresse 42000 ville det se således ud:

Assembler	Decimal kode
JP 42000	195 16 164

Hvorfor hoppe til en anden adresse ?

Der kan være mange grunde til dette. Ofteste skyldes det at man ønsker at genbruge en del af sit program flere gange. Programmet bliver herved mindre og ofte lettere at forstå funktionen af. Programmet bliver mindre (normalt), fordi du ikke behøver at skrive de samme sekvenser at instruktioner flere gange, men at du kan hoppe til det sted hvor du har instruktionerne stående. Du kender formentlig princippet fra BASIC, hvor kommandoen GOSUB netop knytter sig til sådanne mindre genbrugte dele af programmet. Du kender vel derfor også navnet på sådanne dele af programmet - nemlig subrutiner.

Man ser ofte ordet "struktureret programmering" knyttet til det at opdele sine programmer i lutter subrutiner, hvor een central 'stamme' eller hovedrutine kalder samtlige rutiner efter tur. Ved du så hvad hver enkelt rutine udfører vil du relativt let kunne overskue hvad hele programmet laver. Dette betyder at når du laver subrutiner så skal du prøve på at forenkle hver subrutines funktion og omfang. Jo mere simpel dens funktion er - des lettere vil det være at forstå funktionen. Samtidig skal du forsøge at gøre subrutinerne så selvstændige som muligt, d.v.s. mindske overførslen at tal i form af parametre eller andre data til og fra subrutinerne. Dette vil også gøre rutinerne lettere at forstå.

Prøver du at leve op til det ovenfor nævnte kan du let indse hvor vigtigt det er at kunne hoppe til en ny adresse. Hver gang du skal bruge en nye del af programmet skal du bruge en hopinstruktion.

Vi skal nu se lidt mere på forskellen mellem " JR " og " JP ". Som det er nævnt hopper man til adresser med " JP " svarende til kommandoen GOTO med BASIC. Men du kender problemet med at ændre på linienumrene og måske ved et uheld findes linien der skal hoppes til i programmet, fordi du måske har slettet den. Hvis du havde en kommando i BASIC, der tillod at hoppe et antal linier frem ville problemet være løst. Dette er egentlig måden hvorpå " JR " fungerer på.

Lad os se på et lille eksempel. Det nedenfor viste program har vi indlagt i lageradresserne 42000 og frem:

Assembler	Decimal kode
LD BC.32649	1 137 127
OUT (C).C	237 73
LD C.32	14 32
LD A.C	121
CALL 5120	205 0 20
INC C	12
CP 255	254 255
RET Z	200
JP 42007	195 23 164

Figur . Program tiludskrivning af ASCII-karakterer m.m.

Hvis vi kigger i lageradresserne vi nu har lagret med maskinkodeprogrammer ser det således ud:

42000	1	LD BC.32649
42001	137	
42002	127	
42003	237	OUT (C).C
42004	73	
42005	14	LD C.32
42006	32	
42007	121	LD A.C
42008	205	CALL 5120
42009	0	
42010	20	
42011	12	INC C
42012	254	CP 255
42013	255	
42014	200	RET Z
42015	195	JP 42007
42016	23	
42017	164	

Figur . Oversigt over lageradresser.

Programmet indeholder en hopinstruktioner til adresse 42007. Her fortsættes et loop, som gør at vi får udskrevet samtlige symboler mellem 32 og 255. Men hvad vil der ske hvis vi flytter dette program til en anden adresse i maskinens lager ?

Programmet vil naturligvis som før hoppe til adresse 42007, når udførelsen af JP skulle ske. Dette kan meget let føre til et 'crash'.

Løsningen på sådanne problemer kunne være at bruge den relative hop instruktion " JR ". Med denne skal du angive hvor mange bytes, d.v.s. til antal adresser der skal hoppes frem eller tilbage. Da der i programmet skal hoppes til

LD A.C

instruktionen skal vi ændre programmet til

Assembler	Decimal kode
LD BC.32649	1 137 127
OUT (C).C	237 73
LD C.32	14 32
LD A.C	121
CALL 5120	205 0 20
INC C	12
CP 255	254 255
RET Z	200
JR -10	24 246

Figur . Program tiludskrivning af ASCII-karakterer m.m.

Bemærk at de -10 IKKE betegner linier, men bytes. Måden hvorpå du beregner forskydningen sker således

forskydning = 255 - antal bytes + 1

når du skal hoppe baglæns. Du kan maksimalt hoppe 128 bytes bagud. Beregningen skal vi vende tilbage til på næste side.

Den måde vi har løst problemet på gør programmet uafhængig af på hvilken adresse du placerer det. Generelt kaldes programmer, der ikke er afhængig af adressen de placeres på for POSITIONS UAFHÆNGIGE. Vort program er imidlertid ikke sådan idet vi jo udnytter en rutine i operativsystemet som ligger på en bestemt adresse. Flyttes denne fungerer programmet ikke. Man kan imidlertid sige at programmet er positionsuafhængigt på AMSTRAD-maskiner.

Nu skal du imidlertid ikke tro at alt er lettere med " JR " instruktionen. Specielt to ting gør at " JP " alligevel vil blive anvendt meget.

Det første er at hvis du mindsker eller øger antallet af bytes mellem " JR " og adressen der hoppes til. Tilføjer du f.eks. en instruktion eller fjerner en - vil antallet af bytes ændres og hoppet vil blive galt. Dette problem har vi ikke med den anden instruktion " JP ".

Det andet er at du desværre ikke kan hoppe et givet antal bytes frem og tilbage med " JR ". Nedenfor er vist etskema med de adresser der kan hoppes til udfra en given adresse:

Skema . Beregning af forskydning for " JR ".

Adresse	forskydning	Kommentar
RAMTOP		
...		
...		
40000	127	lageradressen er KUN et eksempel,
39999	126	og kunne være en hvilken som helst
39998	125	anden.
39997	124	
...	...	
		0=< n =< 127 = hop fremad
...	...	
...	...	
39875	2	
39874	1	
39873	0	
39872	255	n forskydning
39871	254	JR
39870	253	
39869	252	
...	...	
		128 =< n =< 255 = hop bagud
...	...	
...	...	
39746	129	
39745	128	

Af skemaet fremgår hvordan man beregner forskydningens størrelse når man ønsker at hoppe et givet sted hen.

Når du skal hoppe frem i programmet, d.v.s. opad i lageret mod større adressetal, skal forskydningen være mellem 0 og 127. Ønsker du at hoppe frem til adresse 39995, d.v.s. IKKE adressen men til instruktionen 122 bytes længere fremme.

Som du kan se af figuren over samtlige hopinstruktioner kan hoppene gøres betinget af næsten samtlige flag. Instruktionen til de relative hop kan imidlertid kun gøres betinget af ZERO og CARRY-flaget.

For at illustrere disse instruktioners anvendelse kan vi ændre i programmet vi lavede ovenfor. I dette program brugte vi en RETURNERINGS instruktion for at komme tilbage til BASIC. Instruktionen bruges jo til at kontrollere at vi kommer tilbage til BASIC, når " C "-registret er talt op til 256, hvorved ZERO-flagetjo gøres 'set'. Vi kan afløse denne instruktion med en betinget hop instruktion som f.eks.

JR NZ.byte

og så afslutte programmet med en ubetinget " RET "-instruktion. Dette er vist nedenfor:

Assembler	Decimal kode
LD BC.32649	1 137 127
OUT (C).C	237 73
LD C.32	14 32
LD A.C	121
CALL 5120	205 0 20
INC C	12
CP 255	254 255
JR NZ.-9	24 247
RET	201

Figur .Program til udskrivning af ASCII-karakterer. Version 3.

Sammenligner du denne version med den tidligere vil du se at vi ikke sparer noget ved ændringen.

ZERO-flaget bliver testet og er det 'set' returneres til BASIC via den efterfølgende " RET "-instruktion.

FLAGEFFEKT

Ingen af hopinstruktionerne påvirker flagene i " F "-registret.

DJNZ - INSTRUKTIONEN:

Forkortelsen DJNZ skulle du gerne kunne gennemskue sammensætningen af nu. Den indeholder instruktioner vi har været igennem. D'et står for Decrement (" DEC B ") eller formindsk med een. J'et for Jump (" JR ") for hop og endelig NZ for NOT ZERO eller 'ikke ZERO-flaget'. Instruktionen er sammensat af disse instruktioner:

```
Assembler       'Decimal kode'

...
DEC B           5
JR NZ.byte      32 n
...
```

d.v.s. ialt 2 instruktioner, som normalt ville kræve 3 bytes. Den nye instruktions format er vist nedenfor:

```
Assembler       Decimal kode

DJNZ byte       16 n
```

hvor 'n' er den sædvanlige forskydning, der angiver hvilken vej samt antallet af bytes der skal hoppes. Denne svarer således nøje til forskydningen omtalt i forbindelse med " JR n ".

Den nye instruktion sparer os altså for een byte for hver gang testen skal udføres. Er " B " lagret med tallet 255 sparer vi således 255 bytes ! - hvis vi kan bruge instruktionen.

Forstod ikke helt instruktionens funktion er nedenfor vist hvordan man kunne skabe den samme instruktion blot i BASIC:

```
Linienr.  Kommando

    10    REM ****** DJNZ i BASIC *****
    20    FOR b=x TO 0 STEP -1
    30    GOTO adresse
    40    ...

    90    NEXT b
    95    REM **************************
```

Figur .BASIC-program svarende til DJNZ-instruktionen.

Læg mærke til at 'n' betegner forskydnigen, d.v.s. at du kan hoppe de sædvanlige 127 bytes fremad eller 128 bytes tilbage. Det er derfor vigtigt at huske at 'n' er en enkelt byte og at det ikke skal angives med fortegn. I stedet skal forskydningen overstige de 127 for at kunne angive et hop bagud. Se nedenfor hvordan det er gjort:

Skema . Beregning af forskydning for " DJNZ n ".

Adresse	forskydning	Kommentar
RAMTOP		
...		
...		
40000	127	lageradressen er KUN et eksempel,
39999	126	og kunne være en hvilken som helst
39998	125	anden.
39997	124	
...	...	
		0=< n =< 127 = hop fremad
...	...	
...	...	
39875	2	
39874	1	
39873	0	
39872	255	n forskydning
39871	254	DJNZ
39870	253	
39869	252	
...	...	
		128 =< n =< 255 = hop bagud
...	...	
...	...	
39746	129	
39745	128	

Som nævnt optræder " DJNZ "-instruktionen ofte som del af et loop, hvor instruktionen er tælleren, som kontrollerer antallet af gange loopet skal gennemløbes. Det er " B "-registret, som udgør tælleren. Da " B "-registret er et 8-bit register kan du maksimalt lagre tallet 255 heri. Hermed må du imidlertid IKKE tro at du ikke kan gennemløbe loopet flere end 255 gange. Dette skal vi se på i slutningen af afsnittet.

Det lille loop, d.v.s. hvor loopet ikke gennemløbes mere end 255 er vist nedenfor:

```
            Assembler           Decimal kode

            DI                  243
            LD BC.32649         1 137 127
            OUT (C).C           237 73

            LD B.78             6 78
            LD HL.1831          33 39 7
loop:       LD A.(HL)           126
            CALL 5120           205 0 20
            INC HL              35
            DJNZ loop           16 249
            EI                  251
            RET                 201
```

Figur . Anvendelse af DJNZ til at fremskaffe 'hemmelige' AMSTRAD-meddelelser !.

I dette program benyttes instruktionen til at sikre at loopet ikke gennemløbes mere end de 78 gange. Dette er vigtigt idet AMSTRAD vil crashe, hvis du lader "B" blive væsentlig større, f.eks. 200.

Det du ser på skærmen er ord lagret i din AMSTRADs lager; hvorfor og hvad det betyder må du selv finde ud af. Bagest i bogen er angivet flere adresser, som kan være interessante at kende når du begynder at programmere i maskinkode. Der er de forskellige rutiner vi benytter forklaret.

Den måde hvorpå teksten ses på skærmen skyldes at visse koder sendt til udskrivningsrutinen forårsager linieskift, space, d.v.s. mellemrum, etc. Du kan i din manual se koderne for disse special funktioner. Du kan naturligvis også anvende disse koder i din BASIC-programmering idet AMSTRAD netop bruger samme rutine som vi her har anvendt. Prøv eksempelvis

```
PRINT "dette er linie 1<CTRL>L linie 2"
```

hvor "<CTRL>" betyder et tryk på 'control' tasten. Når du trykker på control og L-tasten skulle du gerne se en nedad pegende pil. Dette er AMSTRADs måde at vise kombinationen af de to taster du har trykket på.

Vi kan, som tidligere nævnt, bruge DJNZ-instruktionen til 'nedtællinger' større end 255. Du kender formentlig hjælpeprogrammer, ofte kaldet toolkits, som gør det muligt at slette, flytte eller kopiere hele blokke af data fra et til et andet sted i lageret. Vi skal i forbindelse med et sådant program benytte os af " DJNZ " som illustration af loopprincippet.

Programmets første del vil vi have til at kunne slette en given blok af data angivet af os ved adresser i lageret. Dette kunne f.eks. være hele eller dele af skærmbilledet. Grunden til at dette er et godt eksempel er at skærmbilledet netop fylder mere end 255 bytes, hvorved vi får brug for at lave vort loop større end de 255. Sletningen af skærmbilledet kunne ske ved at lagre en bestemt værdi i samtlige adresser, der er del af skærmbilledet.

Sletningen sker ved lagring af startadressen i " DE "-registerparret, mens antallet af bytes der skal slettes anbringes i " BC ". I programmet benyttes " HL "-registerparret kun til at lagre det tal, som skal skrives oven i området der skal slettes. Dette tal er i vores eksempel 0.

```
                Assembler       Decimalkode

                LD DE.49152     17 0 192
                LD HL.42000     33 16 164
                LD BC.16384     1 0 64
blloop:         PUSH BC         197
                LD B.255        6 255
                LD (HL).0       54 0
lloop:          LD A.(HL)       126
                LD (DE).A       18
                INC DE          19
                DJNZ lloop      16 251
                POP BC          193
                DJNZ bloop      16 242
                RET             201
```

Figur .Blok-sletningsprogram.

De to 'labels' bloop og lloop er to tæller loops, hvor lloop ligger inde i bloop. Lloop udføres 255 gange for hver gang bloop udfører eet loop. Herved kan vi køre et multiplum af 255 gange med lloop.

Princippet går igen i mange blokkommandoer, f.eks. flyt-blok etc. Der skal imidlertid ikke laves meget om i vores program før dette kunne foretage flytninger i stedet for sletninger. Lad os se på dette.

Et blokflytningsprogram får, som her, behov for information om fra hvilken adresse, der skal flyttes bytes. Angivelse af til hvilken adresse flytningen skal ske er naturligvis også nødvendig - enten i form af en adresse eller et antal bytes, der så skal lægges til eller trækkes fra startadressen. Endelig må vides hvormange bytes der skal flyttes.

En meget brugt fremgangsmåde ved programmering af sådanne opgaver er først at lave programmet i BASIC. I dette tilfælde kunne du med INPUT-kommandoen få adresserne og med POKE og PEEK aflæse og lagre bytene, der skal flyttes. Når dette program kører kunne du begynde at lave din maskinkode programmering. Overføringen af adresser sker lettest med POKE-kommandoen, med mindre du vil ud i større programmer.

Der findes flere og væsentlig bedre metoder at overføre information (parametre) til maskinkodeprogrammer. Du kunne eksempelvis lagre adresserne i en variabel med LET-kommandoen. Overføringen kunne da ske ved at dit maskinkodeprogram søger gennem samtlige variable; finder den rigtige (på navnet f.eks.) og lagrer værdien i det rette register. Dette kræver naturligvis et godt kendskab til hvordan AMSTRAD lagrer tal i variable.

For at få et blokflytnings program må vi ændre vores bloksletnings program et par steder. Hvis vi overfører adresserne til programmet via INPUT- og POKE-kommandoerne må vores program ændres til at kunne aflæse disse adresser. Herefter skal vi teste på hvor mange bytes vi skal flytte. Dette skyldes opbygningen af loopene, hvor det inderste loop jo gennemløbes 255 gange. Er antallet af bytes mindre end 255 skal vi kun udføre det inderste loop antallet af gange vi har bytes der skal flyttes. Endelig skal vi bruge " HL "-registerparret til at aflæse det område, der skal flyttes.

Hvis vi i første omgang tillader kun at flytte den samme blok hver gang programmet udføres, kan vi koncentrere os om det øvrige program forløb. Programmet er vist nedenfor:

	Assembler	Decimalkode
	LD DE.30000	17 48 117
	LD HL.368	33 112 1
	LD BC.255	1 255 0
	XOR A	175
	ADD B	128
	JR Z.lowbyte	40 14
blloop:	PUSH BC	197
	LD B.255	6 255
lloop:	LD A.(HL)	126
	LD (DE).A	18
	INC HL	35
	INC DE	19
	DJNZ lloop	16 250
	RET C	216
	POP BC	193
	DJNZ bloop	16 243
lowbyte:	LD A.0	62 0
	ADD A.C	129
	RET Z	200
	SCF	55
	LD B.A	71
	JR lloop	24 238

Figur .Blok-flytningsprogram.

Ser du bort fra de tre første instruktioner, som alligevel skal ændres i forbindelse med overføring af adresser kan vi starte med at omtale " XOR A "-instruktionen. Denne instruktion sletter indholdet i " A "-registret. Vi blev ovenfor enige om at hvis antallet af bytes der skulle flyttes var mindre end 256, så skulle bloop ikke bruges. Vi tester derfor " B "-registrets størrelse. Dette register må jo være større end nul, hvis antallet af bytes skal overstige 255.

Hvis " B " er mindre end 256 hoppes til lowbyte rutinen, hvorfra der igen hoppes til lloop. Herefter er det kun lowbyte rutinen, der adskiller sig fra vores sletningsprogram. Den lille instruktion " SCF " betyder, som du nok har gættet, gør CARRY-flaget 'set'. Vi bruger dette flag til at signalere at udførelsen har været igennem lowbyte rutinen. På denne måde kan vi bruge instruktionen " RET C ", når loopet skal afsluttes. Dette er en bekvem måde at undgå at der returneres til BASIC, hvis computeren endnu ikke har været igennem lowbyte rutinen.

Dette fører naturligt til en omtale af flageffekten for " DJNZ " instruktionen. Denne er imidlertid ikke omfattende, idet ingen flag påvirkes af instruktionen sålænge " B "-registret ikke er talt ned til 0. Når dette sker gøres ZERO-flaget 'set'. Dette er netop grunden til at vi ovenfor kan anvende dette flag, da dette jo ikke slettes af de øvrige instruktioner.

Vi skal som afslutning på dette afsnit indføre ændringerne i programmet, så vi fra et BASIC-program kan overføre adresserne. Endvidere skal BASIC-programmet skrives, hvilket ikke skulle volde problemer. I stedet for at lagre de tre registerpar " BC ", " DE " og " HL " med konstanter, skal de i stedet lagres med indholdet af et par adresser. Du kan egentlig selv vælge hvilke adresser du vil have brugt.

Vi kan f.eks. lade adresse 41000 til (og med) 41005 indeholde informationen til maskinkodeprogrammet. Dette kunne give dette BASIC-program:

```
Linienr.  kommando

    10    REM *** blok-flytning *********
    20    INPUT "startadresse : ";s
    30    POKE 41001,INT (s/256):
          POKE 41000,(s/256-PEEK(41000))*256
    40    INPUT "destination : ";d
    50    POKE 41003,INT (d/256):
          POKE 41002,(d/256-PEEK(41000))*256
    60    INPUT "antal bytes : ";a
    70    POKE 41005,INT (a/256):
          POKE 41004,(a/256-PEEK(41000))*256
    80    CALL adresse
```

Figur .BASIC-program til overføring af adresser m.m.

Hvis du er bare lidt skrap til BASIC-programmering skulle du kunne reducere programmet en smule med bruge af en subrutine. En andem ting er at det vel var på sin plads at have en linie, der aflæste de nye adresser - blot for kontrollens skyld, eksempelvis en linie som:

```
    90    FOR b=d TO d+a:PRINT b;PEEK (b):NEXT b
```

I linie 80 skal du indsætte adressen fra AMSTRADs lagercelle, hvor maskinkodeprogrammet starter.

Vi mangler nu kun at ændre de tre første instruktioner i maskinkode programmet. Nu skal registreparrene aflæse adresserne 41000 til 41005. Det er imidlertid meget vigtigt at registerene får de rigtige værdier overført. Således SKAL " BC " have antallet af bytes der skal flyttes, d.v.s. indholdet i adresserne 41004 og 41005. " HL " skal have adressen på stedet hvorfra der skal flyttes bytes, d.v.s. indholdet i 41000 og 41001. Endelig skal " DE " have stedet hvortil der skal flyttes (kopieres), d.v.s. indholdet i 41002 og 41003. Dette er gjort nedenfor:

Assembler	Decimal kode
LD HL.(41000)	42 40 160
LD DE.(41002)	237 91 42 160
LD BC.(41004)	237 75 44 160

Figur .Overførsel af adresser m.m. til maskinkode.

Disse tre instruktioner skal udskifte de tre første i det ovenfor viste program. Det er måske på sin plads at erindre dig om at

LD registerpar.(adresse)

udføres ved at lowbyten bliver lagret med indholdet i den angivne adresse, mens highbyten lagres fra adressen+1. Dette er grunden til rækkefølgen i POKE-kommandoerne i BASIC-programmet ovenfor.

Vi har nu lavet et fuldstændigt "blok move" program, og selvom det absolut kunne raffineres er det grundtanken med anvendelsen af DJNZ-instruktionen som du skulle have lært nu.

CALL - INSTRUKTIONEN:

" CALL "-instruktionen svarer nøje til BASIC-koammdoen CALL, som skal efterfølges af en adresse. Vi kan således kalde et program - egentlig en subrutine -, hvorved returadressen gemmes, således at AMSTRAD kan findes tilbage til adressen, hvorfra CALL udførtes. Denne adresse kan vi få at se ved at bruge en af stak-instruktionerne nemlig " POP "- eller en af lagrings-instruktionerne. Lad os f.eks. prøve at finde returadressen til BASIC:

Assembler	Decimal kode
DI	243
LD BC.32649	1 137 127
OUT (C).C	237 73
POP HL	225
LD (41000).HL	34 40 160
PUSH HL	229
EI	251
RET	201

Figur . Kopiering af returadresse til BASIC.

BASIC-programmet er vist nedenfor med CALL-kommandoen efterfulgt af adressen på ovenstående program:

```
Linienr.  Kommando

    10    REM ****** returadresser ***************
    20    FOR a=1 TO 13
    30    READ f
    40    POKE a+41999,f
    50    NEXT a
    60    DATA 243,1,137,127,237,73,225,34,40,160,
          229,251,201
    70    CALL 42000
    80    PRINT PEEK(41000)+256*PEEK(41001)
    90    REM ************************************
```

Figur .BASIC-loader.

Når du har kørt dette program skulle adressen 47514 gerne stå på din computers skærm. Dette er returadressen til BASIC. Hvis du prøver at PEEK'e på disse adresser vil det IKKE være BASIC-området du aflæser. Dette skal gøres med maskinkode - f.eks. med et program som det nedenstående:

Assembler	Decimal kode
DI	243
LD BC.32649	1 137 127
OUT (C).C	237 73
LD HL.49152	33 0 192
LD DE.20000	17 32 78
LD BC.16384	1 0 64
LDIR	237 176
EI	251
RET	201

Figur .Maskinkode program til 'aflæsning' af AMSTRADs BASIC.

Når dette program er udført kan du aflæse hele AMSTRADs BASIC-område ved at PEEK'e fra adresse 20000 og 16 Kilo bytes frem. Skal du finde adresse 47514 skal du således fratrække 20001 fra adressen og PEEK'e på denne adresse. Da bogen her ikke er en disassemblering af AMSTRADs ROM må du give dig på vej alene. En disassemblering ville kræve en hel bog for sig og det er helt sikkert at det ville lære dig en hel del om AMSTRADs funktionsmåde at foretage sådanne disassembleringer.

Udførelsen af " CALL "-instruktionen kan også gøres afhængig af flag som det var tilfældet med hopinstruktionerne m.fl. Du kan eksempelvis betinge kaldet af om ZERO-flaget i " F "-registret er "set". Dette skrives:

CALL Z.adresse

hvor du som normalt udskifter 'adresse' med tallet du skal hoppe til. Skulle du ikke kunne huske hvad der gør ZERO-flaget "set", så se eventuelt i Appendix for oversigten af flageffekterne.

De fire mest brugte flag kan bruges sammen med " CALL "-instruktionen. Disse er vist nedenfor:

1. CARRY	a:	om CARRY er "set" =	C
	b:	om CARRY er "reset" =	NC
2. ZERO	a:	om ZERO er "set" =	Z
	b:	om ZERO er "reset" =	NZ
3. SIGN	a:	om SIGN er "set" =	M
	b:	om SIGN er "reset" =	P
4. PARITET	a:	om PARITET er "set" =	PE
	b:	om PARITET er "reset" =	PO

Længst til højre i skemaet kan du se de forkortelser, som man bruger i forbindelse med instruktionsnavnet. Skriver du således

```
CALL P.30000
```

vil AMSTRADs mikroprocessor teste SIGN-flaget. Hvis SIGN-flaget er "reset" udføres CALL-instruktionen ellers hoppes til næste instruktion umiddelbart efter CALL.

Det kan godt være lidt vanskeligt at huske disse forkortelser specielt da de jo er engelske. Nedenfor er opstillet de relevante spørgsmål til de forskellige flag:

```
 C : CARRY testes for "set"        : er der mente ?
NC : CARRY testes for "reset"      : er der ikke mente ?

 Z : ZERO testes for "set"         : er resultatet nul ?
NZ : ZERO testes for "reset"       : er resultatet ikke nul ?

 M : SIGN testes for "set"         : er resultatet negativt ?
 P : SIGN testes for "reset"       : er resultatet positivt ?

PE : PARITET testes for "set"      : er der lige paritet ?
PO : PARITET testes for "reset"    : er der ulige paritet ?
```

Som betingelserne her er stillet op vil et svar lig med:

JA : betyde at instruktionen udføres.
NEJ: betyde at AMSTRAD udfører instruktionen der kommer lige efter.

Som du nok kan tænke dig står " N "'et for " not ... " eller sagt på anden måde, at flaget IKKE antager en eller anden værdi.

" CALL "-instruktionen fungerer sammen med instruktionen " RET ", som gennemgåes efter dette afsnit. Idet " RET " er forkortelsen for RETURN kan du vel se ligheden til GOSUB/RETURN-kommandoerne fra BASIC. De fungerer på nøjagtig samme måde. Dette betyder at hvis du udfører en " CALL "-instruktion, så vil du kunne komme tilbage til stedet (+2) hvor CALL-instruktionen står, når du udfører en RET-instruktion. Dette er imidlertid ikke hele sandheden som du vel ved fra BASIC. Bruger vi eksempelvis en " CALL "-instruktion i en af vores maskinkodeprogrammer, som vi senere skal, vil AMSTRAD ikke returnere til BASIC ved mødet med den første " RET "-instruktion. I stedet vil den returnere tilbage til adresse lige efter " CALL ". Anden gang den møder en " RET " sker returneringen, hvis du ikke i mellemtiden har fået udført nye " CALL "-instruktioner. Som med BASICs GOSUB/RETURN skal du udvise streng symmetri med " CALL "/" RET " for ikke at komme galt af sted.

Når " CALL " udføres gemmes den adresse som står hvori instruktonen står, d.v.s. den adresse hvori " CALL " står. Denne skubbes ind på stakken svarende til at vi brugte " PUSH "-instruktionen. Herefter læses adressen i CALL-instruktionen ind i Program Counter registeret og dette forårsager AMSTRAD at hoppe til denne adresse. herfra fortsættes udførelsen. Du kan herfra tage returadressen af stakken med en " POP "-instruktion eller du kan lade den stå. Hvis du ønsker at returnere til CALL-adressen igen må du lade den ligge på stakken. Når du vil returnere udfører du blot en " RET "-instruktion. Denne tager adressen af stakken og lægger den i Program Counteren og AMSTRAD hopper tilbage. Men IKKE præcis til samme adresse. AMSTRAD lægger to til adressen, hvilket netop svarer til første adresse efter CALL-instruktionen.

" RET "-instruktionen kan imidlertid ikke selv finde den rigtige adresse på stakken. Instruktionen tager de to bytes som stakpegerregistret peger på. Disse to tal omformes til en adresse og denne hoppes der til. Kan du nu se hvad der vil ske, hvis du har brugt stakken til at lagre tal på i mellemtiden ????. Dette kommer vi tilbage til ved gennemgangen af " RET "-instruktionen i næste afsnit.

Der findes 9 " CALL "-instruktioner, som alle er vist nedenfor:

Assembler	Decimal kode
CALL adresse	205 n n
CALL C.adresse	220 n n
CALL NC.adresse	212 n n
CALL Z.adresse	204 n n
CALL NZ.adresse	196 n n
CALL M.adresse	252 n n
CALL P.adresse	244 n n
CALL PE.adresse	236 n n
CALL PO.adresse	228 n n

Figur . Oversigt over " CALL "-instruktionen.

Adresseangivelserne må du efterhånden kende. Først 'n' er lowbyten i adressen, og anden highbyten, eksempelvis

CALL 65000 205 232 253

idet du kan kontrollere low- og highbytene med 232+253*256.

Du har nu mulighed for at kalde samtlige rutiner i AMSTRADs operativsystem samt BASIC-fortolker. Du skal blot kende de rette adresser. Flere af disse er vist i slutningen af bogen, hvor flere udskrivningsrutiner og andre skærmbehandlingsrutiner er vist adresserne til. Ofte skal registrene også 'forberedes' inden du kalder rutinerne.

Når du tænder for din computer udføres faktisk hvad der svarer til instruktionen:

CALL 0

d.v.s. et hop til adresse 0. Herved slettes alt i maskinens lager og den gøres klar til at modtage dine indtastninger. Vi kan afprøve dette med nedenstående:

```
10 POKE 20000,205
20 POKE 20001,0
30 POKE 20002,0
40 CALL 20000
```

Når du udfører dette program skal du ikke have noget vigtigt i AMSTRADs lager, idet alt vil blive slettet i maskinen.

Skal du hoppe til en given adresse, f.eks. i AMSTRADs ROM kan dette gøres med både JUMP og CALL. Der er imidlertid den vigtige forskel, som ikke må glemmes - nemlig at CALL hænger sammen med RET. Dette betyder at når AMSTRAD møder en RET vil den hoppe to forskellige steder hen afhængig af hvilken instruktion som har kaldt rutinen med RET. Vi kan illustrere dette ved at hoppe til en rutine i ROM'en med både " CALL " og " JP ". På adresse 1683 i AMSTRADs ROM begynder copyright meddelelsen du ser på skærmen hver gang du tænder for maskinen. Vi kan få denne frem ved at kalde udskrivningsrutinen med angivelse af hvor det vi vil have udskrevet angives i " HL "-registerparret. Dette er vist nedenfor:

assembler	decimal kode
DI	243
LD BC.32649	1 137 127
OUT (C).C	237 73
LD HL.1683	33 147 6
CALL 1771	205 235 6
EI	251
RET	201

Figur .Udprintning af copyright meddelelsen.

Prøver du nu at udskifte " CALL " med " JUMP " vil de se forskellen.

RET - INSTRUKTIONEN:

Vi har flere gange tidligere i bogen brugt instruktionen " RET " som jo er forkortelsen for -return- eller hop tilbage. Vi brugte den da til at komme tilbage til BASIC.

Der findes mere end denne ene udgave af " RET ". De øvrige 8 instruktioner bruges sammen med en test af een af de 4 mest brugte flag i " F "-registret. Det drejer sig om:

1. CARRY
2. ZERO
3. SIGN og
4. PARITETS-flagene.

Som tidligere kaldes disse instruktioner for 'betingede'.

Når en " CALL "-instruktion udføres vil adressen lige efter CALL-instruktionen blive lagret på stakken. Eksempelvis vil

adresse	indhold	instruktion
10000	...	
9999	1	
9998	20	
9997	205	CALL 276 = 20+1*256
9996	...	

stakken efter den viste " CALL "-instruktion være lagret med adressen 10000. Vi kan teste dette med nedenstående program, som netop aflæser indholdet på stakken:

Adresse	Assembler	Decimal kode
10000	CALL 10004	205 20 39
10003	RET	201
10004	POP HL	225
10005	LD (40000).HL	34 64 156
10008	PUSH HL	229
10009	RET	201

Det er vigtig at du lægger programmet på de angivne adresser. Når du har fået programmet udført skal du med PEEK aflæse adresserne 40000 og 40001, som er henholdsvis low- og high byte i den returadresser vi har aflæst på stakken. Denne skulle gerne være 10003. Dette stemmer med det vi omtalte ovenfor, nemlig at den adresse hvorfra udførelsen skal ske lagres på stakken.

En lille biting vil være at overveje hvad der ville ske ved at fjerne næstsidste instruktion, d.v.s. " PUSH HL ".

Når " RET "-instruktionen udføres trækkes 'den nederste' adresse af stakken, hvorved stakken gøre to bytes mindre. Disse to bytes er nu returadressen.

Med det ovenfor viste testprogram skulle du have indset at det er muligt at ændre indholdet af stakken, d.v.s. vi kan ændre de adresser der ligger herpå. Gør vi dette vil de forskellige " RET "-instruktioner få AMSTRAD til at hoppe til andre end de planlagte steder. Vi kan således hoppe rundt i AMSTRADs lager uden at bruge hverken JUMP eller CALL instruktioner. Nedenfor er vist et eksempel på dette:

```
                Assembler       Decimal kode

                DI              243
                LD BC.32649     1 137 127
                OUT (C).C       237 73
                LD HL.1645      33 109 6
                LD DE.1771      205 235 6
                LD BC.return    1 34 39
                PUSH BC         197
                PUSH DE         213
                RET             201

    return:     EI              251
                RET             201
```

Figur . Illustration af " RET "-instruktionens funktion.

Hvis du ikke vil ændre i programmet er det vigtigt at du lagrer dette på adresse 10000 og fremefter. Dette skyldes den absolutte adresse der lagres i " BC "-registerparret i 6. sidste instruktion. Vil du flytte programmet skal denne adresse ændres tilsvarende, nemlig til programstart plus 18 bytes.

I stedet for at bruge " CALL " til at kalde udskrivningsrutinen kan vi som vist lige så godt bruge " RET ". Vi skal blot huske at lægge endnu en returadresse på stakken, hvis vi ønsker at få programkontrol over AMSTRAD igen inden vi returnerer til BASIC.

Der findes ialt 9 returnerings-instruktioner, hvoraf de 8 er betingede. De er ikke absolut nødvendige, idet de kunne erstattes af en kombination af f.eks. en betinget " JR " og den ubetingede " RET ". Dette er vist under oversigten for instruktionerne:

Assembler	Decimal kode	Kommentar
RET	201	returner til CALL.
RET C	216	--,når CARRY er "set".
RET NC	208	--,når CARRY er "reset".
RET Z	200	--,når ZERO er "set".
RET NZ	192	--,når ZERO er "reset".
RET M	248	--,når SIGN er "set".
RET P	240	--,når SIGN er "reset".
RET PE	232	--,når PARITET er "set".
RET PO	224	--,når -- er "reset".

Figur . Oversigt over " RET "-instruktionerne.

Et program der kunne erstatte " RET C "-instruktionen er vist nedenfor:

```
          Assembler       Decimal kode

          ...
          JR C.exit       56 2
          JR loop         24 n
exit:     RET             201
```

Det må imidlertid være klart, at denne erstatning kræver flere bytes end den ene, som " RET C " bruger. De betingede returnerings-instruktioner er således værdifulde.

Et andet lille eksempel der kan illustrere brugen af returnerings-instruktionerne ses nedenfor. Programmet kunne kaldes et bloksøgnings program, idet det foretager en søgning gennem en blok af adresser efter en bestemt byte, f.eks. et tal. Vi benytter her den automatiske sammenligningsinstruktion " CPIR ", som blev gennemgået tidligere.

Programmet indledes med at vi lagrer " A "-registret med det tal vi ønsker at søge efter. " BC "-registerparret lagres med det antal adresser vi vil gennemsøge og endelig lagres " HL " med den adresse, hvorfra vi ønsker at indlede søgningen. Selve søgningen sker ved at " HL " peger på den adresse, som skal testes op imod " A "-registret. Er indholdet i adressen lig med indholdet i " A "-registret er vi færdige og skal blot gennem adressen til senere brug. Er de ikke ens skal " BC " formindskes med een og " HL "forøges med een. Processen går derefter om. Dette vil fortsætte indtil tallet er fundet eller at " BC "-registerparret når nul.

Assembler	Decimal kode
LD HL.49152	33 0 192
LD BC.16384	1 0 64
LD A.255	62 255
CPIR	237 177
LD (10000).HL	34 16 39
RET NZ	192
DEC HL	43
LD (10000).HL	34 16 39
RET	201

Figur . Bloksøgningsprogram.

Arsagen til at vi starter i adresse 49152 skyldes at her starter skærmbillede mappen. Du kan naturligvis ændre denne samt de øvrige adresser efter ønske.

" HL " lagres i adresse 10000, så du senere kan finde adressen hvor tallet blev fundet - HVIS tallet blev fundet. Dette kan du KUN afgøre ved at aflæse adresse 10000. Er dette tal lig med startadressen plus antal gennemsøgte bytes plus 1 - fandt AMSTRAD ikke tallet. Ellers findes tallet i adresser på adresse 10000 og 10001.

Arsagen til at " HL " må reduceres med een, hvis den søgte byte findes skyldes " CPIR "-instruktionen funktionsmåde. I denne øges " HL " først, FØR sammenligningen mellem indhold i adressen og " A "-registret.

Blot for at illustrere hastighedsfordelen ved programmering i maskinkode så prøv at udføre nedenstående program, der udfører nøjagtig det samme.

```
Linienr. Kommando

    10   REM ****** blok søgningsprogram *****
    20   FOR a=49152 TO 65535
    30   IF PEEK(a)=255 THEN PRINT a: END
    40   NEXT a
    50   ************************************
```

Figur . Bloksøgningsprogram i BASIC.

Som du kan se er der overhovedet ingen sammenligning mellem de to typer programmering.

FLAGEFFEKT

Instruktionerne i denne gruppe påvirker IKKE flagene.

ROTATIONS / SKIFT - INSTRUKTIONERNE

Navnet skal betyde,at der flyttes (roteres og skiftes) om på tallene i byten,d.v.s. at de 1 og 0'er,som udgør byten flyttes een plads mod venstre eller højre med instruktionerne i denne gruppe.

Dette er nogle uhyre nyttige instruktioner,såfremt man kender lidt til de forskellige regneregler,der knytter sig til sådanne flytten om på binære tals pladser.

Flyttes samtlige bit een plads mod venstre,d.v.s. at bit nr. 3 flyttes til bit nr. 4's plads osv. gøres tallet i byten præcis dobbelt så stort.Dette forudsætter dog,at ingen bit går tabt,og i dette tilfælde gælder det ydertilfældene bit nr. 7.Bit nr. 0 skal være "reset",efter den oprindelige bit flyttes til bit nr. 1's plads.

Dette kan være lidt vanskeligt at forestille sig,så du kan se situationen i figur 1.

```
oprindelige byte (71):  0100 0111
efter flytning  (142):  1000 1110
                        -------
```

Figur 1. Flytning af bit een plads mod venstre.

Husk,når du ser opstillingen at bytens 8 bit nummeres bagfra.Største bit er således nr. 7.Du husker forhåbentlig beregningen,som kan omsætte binære tal til decimale:

$$71: \quad 0*2^7 + 1*2^6 + 0*2^5 + 0*2^4 + 0*2^3 + 1*2^2 + 1*2^1 + 1*2^0$$
$$= \quad 64 + \quad 4 + 2 + 1$$
$$= 71$$

$$142: \quad 1*2^7 + 0*2^6 + 0*2^5 + 0*2^4 + 1*2^3 + 1*2^2 + 1*2^1 + 0*2^0$$
$$= 128 + \quad 8 + 4 + 2$$
$$= 142$$

Vi kunne selvfølgelig have valgt et hvilket som helst

andet tal,blot vi overholdt forudsætningen nævnt i starten.

Hvad er mere naturligt end når et tal fordobles ved at flytte samtlige bit een plads mod venstre at tallet halveres hvis man rykker samtlige bit een plads mod højre.Dette er også tilfældet.

Vi må igen tage forbehold for at (denne gang) bit nr. 0 ikke forsvinder.Dette betyder,at hvis bit nr. 0 (før bit nr. 7) er " set ",d.v.s. 1,kan teknikken volde os lidt problemer.Vi kan imidlertid bruge en af de logiske instruktioner for at "reset" f.eks. bit nr. 0.Dette er bare ikke den bedste løsning af verdenen,idet vi ikke får besked om bitten ER " set " eller.Vi må altså teste den først.Når det gælder bit nr. 0 kan vi bruge f.eks. "ENKELT BIT INSTRUKTIONERNE",som bliver gennemgået senere til testen.Her kan hver enkelt bit testes og styres.Vi kan så bruge flagene,og aflæse disse for informationen.Det samme kan lade sig gøre med bit nr. 7.Disse tests skal selvfølgelig foretages før flytningen.

Det er disse flytninger med bit,der foretages med instruktionerne i denne gruppe.I mange tilfælde inddrages også carry-flaget i " F "-registret på lige fod med de øvrige bit, således at en flytning mod venstre ofte vil forårsage at bit nr. 7 flyttes til carry,carrys gamle bit til bit nr. 0 osv. Hermed er problemet med den tabte bit imidlertid ikke ovre selvom vi har fået lavet en ring,som bevarer alle bit.

Instruktionerne er inddelt i 7 mindre grupper.

1. RLC og RLCA instruktionerne,
2. RRC og RRCA ----- ,
3. RL og RLA ----- ,
4. RR og RRA ----- ,
5. SLA ----- ,
6. SRA ----- og
7. SRL instruktionerne.

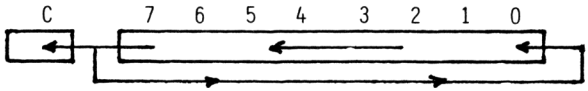
1. " RLC " OG "RLCA" INSTRUKTIONERNE.

Instruktionerne betyder:

RLC : roter mod venstre med carry
RLCA : roter mod venstre med carry,men KUN med " A "-registret.

De to instruktioner er identiske bortset fra at " RLCA " kun kan bruges med " A "-registrets bit,mens " RLC " kan udføres på samtlige enkelt byte registre.

Figur 2 viser skematisk hvad der sker i det registers enkelte bit,som instruktionen bliver brugt på:



Figur 2. " RLC " og " RLCA " instruktionens flytninger.

Hver bit flyttes een plads i pilens retning.Bit nr. 7 flyttes både over i bit nr. 0 samt i carry.Hermed mistes carry flagets bit,idet denne overskrives af bit nr. 7,og kopieres ikke andetsteds.

" RLCA " er som nævnt en special udgave af " RLC ",idet instruktionen (som kræver mindre bytes end " RLC ") kun flytter om på " A "s bit.Der findes med andre ord ikke en " RLCB " etc.

De 10 instruktioner ses i figur 3.

assembler	decimal kode
RLC A	203 7
RLC B	203 0
RLC C	203 1
RLC D	203 2
RLC E	203 3
RLC H	203 4
RLC L	203 5
RLC (HL)	203 6
RLC (IX+n)	221 203 n 6
RLC (IY+n)	253 203 n 6
RLCA	7

Læg mærke til at " RLCA " kun kræver een byte,hvor de øvrige kræver to og tre bytes.

2. " RRC " og " RRCA " INSTRUKTIONERNE.

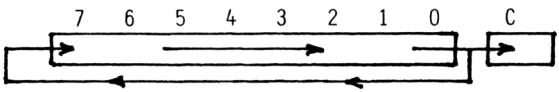
Instruktionerne betyder:

RRC : roter mod højre med carry
RRCA : roter mod højre med carry,men KUN med " A "-registret.

Som du kan se flytter disse instruktioner bittene den modsatte vej af " RLC ".

Igen er de to instruktioner ens,men hvor " RRCA " kun kan rotere " A "-registrets bit.

Figur 3. viser skematisk,hvad der sker i det registers enkelte bit,som instruktionen bliver brugt på.



Figur 3. " RRC " og " RRCA " instruktionernes flytning.

Læg mærke til at carry er flyttet ned i den anden ende af hvad den var i figur 2.Dette skyldes selvfølgelig alene at figuren er lettere at overskue på denne måde.

Igen flyttes hver bit een plads i pilens retning.Den 0'te bit (bit nr. 0) kopieres i både carry samt i bit nr.7. Igen går carry-bitten tabt,idet den jo ikke kopieres til et andet sted.

De 10 instruktioner er

assembler	decimal kode
RRC A	203 15
RRC B	203 8
RRC C	203 9
RRC D	203 10

assembler	decimal kode
RRC E	203 11
RRC H	203 12
RRC L	203 13
RRC (HL)	203 14
RRC (IX+n)	221 203 n 14
RRC (IY+n)	253 203 n 14
RRCA	15

3. " RL " OG " RLA " INSTRUKTIONERNE

Instruktionerne betyder:

RL : roter mod venstre
RLA : roter mod venstre,men KUN med " A "-registret.

Disse instruktioner foretager en 'fuld' rotation mod venstre,hvorved ALLE bit bevares,men blot flyttes een plads mod venstre.Bit nr. 7 går til carry,carry til bit nr. 0 osv.

Figur 4 viser i skematisk form hvad der sker:

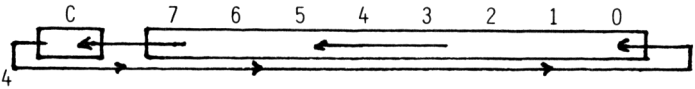


FIG. 4

Alle bit rykker een plads i pilens retning,hvorved samtlige bit bibeholdes uden tab af nogen.

Vi behøver vel næppe nu forklare at en af instruktionerne kun kan bruges på " A "-registret,men så samtidig er byte besparende.

De 10 instruktioner er vist i figur 5

assembler	decimal kode
RL A	203 23
RL B	203 16
RL C	203 17
RL D	203 18
RL E	203 19

assembler	decimalkode
RL H	203 20
RL L	203 21
RL (HL)	203 22
RL (IX+n)	221 203 n 22
RL (IY+n)	253 203 n 22
RLA	23

Figur 5. " RL " og " RLA " instruktionerne.

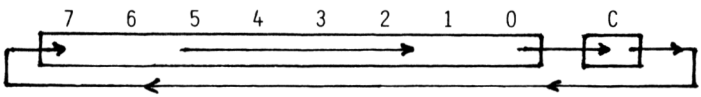
4. " RR " OG " RRA " INSTRUKTIONERNE:

Instruktionerne betyder:

RR : roter mod højre
RRA : roter mod højre,men KUN med " A "-registret.

" RR " foretager en 'fuld' rotation mod højre,hvorved alle bit bevares,men blot flyttes een plads mod højre.Bit nr. 0 kopieres over i carry,carry over i bit nr. 7,osv.

Figur 6 viser,hvorledes bittene flyttes,i det angivne register.



Figur 6. " RR " og " RRA " instruktionernes flytning af bit.

Samtlige bit rykker een plads i pilens retning.

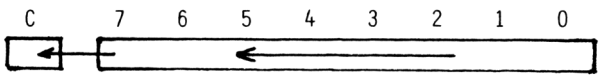
Der findes også her 10 instruktioner,een for hvert enkelt byte register,samt 3 måder at angive en memory byte,hvis indhold skal foretage flytningen.Instruktionerne er vist på næste side.

assembler	decimal kode
RR A	203 31
RR B	203 24
RR C	203 25
RR D	203 26
RR E	203 27
RR H	203 28
RR L	203 29
RR (HL)	203 30
RR (IX+n)	221 203 n 30
RR (IY+n)	253 203 n 30
RRA	31

Figur 7. " RR " og " RRA " instruktionerne.

5. " SLA " INSTRUKTIONERNE:

SLA betyder -skift mod venstre- og skal mene at " SLA " IKKE er en rotation,idet der ikke flyttes nogen bit 'imod' bevægelsen.Hermed menes at den streg,som hidtil har været på de tidligere figurer ikke findes her.Hermed rykker alle bit een plads mod venstre,MEN der overføres IKKE bit fra carry til bit nr. 0.Denne bit forbliver uændret,hvad den er.Der skrives med andre ord ikke i denne bit.Figur 8 viser flytningen af bittene.



Figur 8. Flytningen af bit med skift instruktionen " SLA ".

Igen flytter alle bit een plads i retning med pilen.

Der findes 10 instruktioner,som det ses i figur 9.

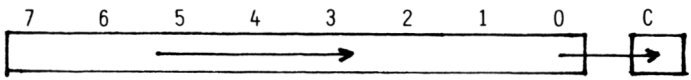
assembler	decimal kode
SLA A	203 39
SLA B	203 32
SLA C	203 33
SLA D	203 34
SLA E	203 35
SLA H	203 36
SLA L	203 37
SLA (HL)	203 38
SLA (IX+n)	221 203 n 38
SLA (IY+n)	253 203 n 38

Figur 9. " SLA " instruktionen.

6. " SRA " INSTRUKTIONEN:

" SRA " er forkortelsen for skift mod højre.Instruktionen flytter alle bit een plads mod højre,og som det er tilfældet med " SLA " kopieres ingen bit fra carry til bit nr. 7.Denne bit forbliver uændret hvad den har været før instruktionen.

Figur 10 viser flytningen.



Figur 10. Flytningen af bit med " SRA " instruktionen.

De 10 instruktioner er

assembler	decimal kode
SRA A	203 47
SRA B	203 40
SRA C	203 41
SRA D	203 42
SRA E	203 43
SRA H	203 44
SRA L	203 45

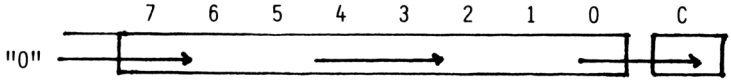
assembler	decimal kode
SRA (HL)	203 46
SRA (IX+n)	221 203 n 46
SRA (IY+n)	253 203 n 46

Figur 11. " SRA " instruktionerne.

7. " SRL " INSTRUKTIONEN:

Denne instruktion fungerer næsten,som " SRA ",men den eneste forskel er,at bit. nr. 7 ikke forbliver uændret,men bliver " reset ".SRL betyder roter mod højre.

Figur 12 viser bittenes flytning.



Figur 12. Skift af bit med instruktionen " SRL ".

De 10 instruktioner er

assembler	decimal kode
SRL A	203 63
SRL B	203 56
SRL C	203 57
SRL D	203 58
SRL E	203 59
SRL H	203 60
SRL L	203 61
SRL (HL)	203 62
SRL (IX+n)	221 203 n 62
SRL (IY+n)	253 203 n 62

Figur 13. " SRL " instruktionen.

Som du kan se i oversigten i APPENDIX D påvirkes samtlige flag af alle rotations og skift instruktionerne, bortset fra de 4 der er specielt tilegnet " A "-registret.

Det drejer sig således om " RLCA "," RRCA "," RLA " og " RRA ", som IKKE påvirker flagene.De øvrige gør dette efter de vanlige regler for flagene.

MULTIPLIKATION OG DIVISION:

Som nævnt i forbindelse med " ADD "-instruktionen kan multiplikation og division udføres hurtigere med de netop gennemgåede instruktioner til skifts og rotationer. Nedenfor skal vi se på nogle programmer til multiplikation af to 8-bit tal og to 16-bit tal samt division af et 8-bit tal op i et 16-bit tal. Først multiplikationen:

MULTIPLIKATION 8-bit * 8-bit:

Programmet er vist nedenfor:

```
          Assembler        Decimal kode

          LD BC.(adr-1)    237 75 64 156
          LD B.8           6 8
          LD DE.(adr-2)    237 91 65 156
          LD D.0           22 0
          LD HL.0          33 0 0

multi:    SRL C            203 57
          JR NC.undlad     48 2
          ADD HL.DE        237 90
undlad    SLA E            203 35
          RL D             203 18
          DEC B            5
          JR NZ.multi      32 243
          LD (adr-3).HL    34 66 156
```

Figur . 8-bit multiplikation.

Den første gruppe af instruktioner forbereder egentlig kun multiplikationen. De to tal der skal ganges med hinanden er lagret i nedenstående adresser:

```
                adresse  indhold

          adr-1   40000  1. faktor
          adr-2   40001  2. faktor (multiplikand)

          adr-3   40002  resultat
                  40003  resultat
```

Årsagen til at highbyte delen af såvel " BC " som " DE " lagres to gange er at de to 8-bit tal jo kun optager lowbytedelen af registerparrene. " B " lagres med en tæller, nemlig 8 idet vi bruger en binær multiplikation, hvor vi skal samtlige 8 bits igennem før vi er færdige.

Selve multiplikationen sker ved at vi først skifter " C "-registret mod venstre, hvorved CARRY vil blive " set ", hvis bit 7 i registret var "set" før skiftet. Denne proces udføres 8 gange, d.v.s. een gang for hver bit. For hver bit der er "set" i " C "-registret adderes indholdet i " DE " til indholdet i " HL ", som egentlig er resultatregistret. Hoppet uden om " ADD "-instruktionen sker således når CARRY ikke bliver "set" som følge af skiftinstruktionen. Hermed er det dog ikke slut. Dette skyldes jo at der er forskel på om bit nr. 1 og bit nr. 4 er "set". Vi bliver derfor nødt til at skifte " DE "-registret for hver gang vi skifter bit i " C "-registret. Da værdien af at bit nr. 2 er "set" netop er dobbelt så stor som hvis bit nr. 1 var "set" skal vi kun skifte " E "-registret een gang.

De sidste instruktioner udfører kun test af at loopet gennemløbes 8 gange.

MULTIPLIKATION 16-bit * 16-bit:

Princippet for denne multiplikation er stort set det samme som ovenfor trods den lidt anderledes forberedelse til multiplikationen:

```
                Assembler        Decimal kode

                LD A.(adr-1h)    58 65 156
                LD C.A           79
                LD A.(adr-1l)    58 64 156
                LD B.16          6 16
                LD DE.(adr-2)    237 91 66 156
                LD HL.0          33 0 0

mult:           SRL C            203 57
                RRA              31
                JR NC.undlad     48 2
                ADD HL.DE        237 90
undlad:         EX DE.HL         235
                ADD HL.HL        237 106
                EX DE.HL         235
                DJNZ mult        16 243
                LD (adr-3).HL    34 68 156
                RET              201
```

Figur . 16-bit multiplikation.

Den vigtigste forskel er vel blot at der ikke findes en instruktion til at skifte hele " DE "-registerparret med, hvorfor vi bliver nødt til først at flytte indholdet over i " HL ", for at udføre 'skiftet mod venstre' i dette registerpar. Dette sker med fordoblingen af " HL ". Herefter skiftes tilbage igen.

Nedenfor er vist hvad de forskellige adresser står for:

```
                adresse  indhold

adr-1           40000    1. faktor low.
                40001      ----     high.
adr-2           40002    2. faktor low.
                40003      ----     high.

adr-3           40004    resultat
                40005    resultat
```

DIVISION 16-bit / 8-bit :

De adresser du skal bruge til lagring af divisor og dividend er vist nedenfor:

	adresse	indhold
adr-1	40000	divisor
adr-2	40001	dividend low.
	40002	---- high.
adr-3	40003	resultat
	40004	resultat

Programmet ses nedenfor hvor første blok igen er forberedelsen til divisionen:

```
            Assembler        Decimal kode

start:      LD A.(adr-1)     58 64 156
            LD D.A           87
            LD E.0           30 0
            LD HL.(adr-2)    34 65 156
            LD B.8           6 8

division    XOR A            175
            SBC HL.DE        237 82
            INC HL           35
            JP P.undlad      242 n n  (start+21)
            ADD HL.DE        25
            DEC HL           43
undlad      ADD HL.HL        41
            DJNZ division    16 244
            RET
```

Figur . Division af 16-bit med 8-bit tal.

Kvotienten lagres i " HL "-registret.

Selve divisionen sker ved fratrækningen af " DE " fra " HL ", hvorefter resultatet (resten) testes. Er den positiv eller nul er divisionen gået godt. Dette sker i JUMP-instruktionen. Samme skift problmatik som ved 16-bit multiplikation.

ENKELT BIT INSTRUKTIONERNE:

De tidligere gennemgåede instruktioner omfattede alle funktioner på enten 8 eller alle 16 bit under eet. Instruktionerne i denne gruppe kan imidlertid anvendes på register- og lageradressers indholds enkelte bit. Man kan således adressere (henvende sig til) samtlige enkelte bit i alle registre og i hele lageret.

Med de tre nedenstående instruktioner:

1. BIT
2. SET
3. RES

kan vi teste, "set"te eller "reset"te samtlige enkeltbit ved angivelse af bittens nr.

1. BIT INSTRUKTIONEN:

Denne instruktion tester om en enkelt bit er "set" eller "reset". Dette gøres ved at aflæse ZERO-flaget, som påvirkes af " BIT "-instruktionen.

Når testen udføres og den testede enkelt bit er:

"set" : bliver ZERO-flaget "reset".

"reset": bliver ZERO-flaget "set".

Som du kan se svarer testen af bitten til spørgsmålet " Er den testede bit nul ? ". Er svaret ja, d.v.s. bitten er "reset", bliver ZERO-flaget "set".

Der findes ialt 64 instruktioner, een for hver bit i hvert 8-bit register, d.v.s. 8*8=64. Herudover kan du via den indirekte adressering, d.v.s. hvor et registerpar indeholder en adresse, som påvirkes, påvirke en lageradresse. Instruktionerne er vist nedenfor:

Bit nr. / Register	0.	1.	2.	3.	4.	5.	6.	7.
A	71	79	87	95	103	111	119	127
B	64	72	80	88	96	104	112	120
C	65	73	81	89	97	105	113	121
D	66	74	82	90	98	106	114	122
E	67	75	83	91	99	107	115	123
H	68	76	84	92	100	108	116	124
L	69	77	85	93	101	109	117	125
(HL)	70	78	86	94	102	110	118	126

Figur. Oversigt over "BIT"-instruktionerne.

De sidste to instruktionstyper skal foranstilles henholdsvis IX med 221 203 n, og IY 253 203 n, hvor 'n' som sædvanlig angiver forskydningen fra basisadressen:

(IX+n)	70	78	86	94	102	110	118	126
(IY+n)	70	78	86	94	102	110	118	126

Bemærk at de to sidste instruktioner har samme koder som testning af (HL), bortset fra de 3 foranstillede koder.

Vi kan illustrere brugen at " BIT "-instruktionen i et program som kan omsætte decimale tal lagret i AMSTRAD til binære tal og udskrive dem på skærmen. Nedenfor i figuren er programmet vist.

Programmet aflæser en lageradresse, hvis indhold er tallet der skal omsættes. Vi lagrer " A "-registret med indholdet i denne adresse og overfører det til " D "-registret. " B "-registret udgør sammen med " DJNZ "-instruktionen en tæller, der sørger for at et loop kun gennemløbes 8 gange, svarende til antallet af bits i " D "-registret. Selve udskrivningen sker ved at vi tester fra den ene ende af " D "-registret hvilke bits der er "set". Disse forårsager udskrivningen af et 1-tal på skærmen, mens en "reset" bit udskrives som et 0.

```
                Assembler          Decimal kode

                LD A.(adresse)     58 0 25
                LD D.A             87
                LD B.8             6 8

loop:           BIT 7.D            203 122
                LD A.0             62 48
                JR Z.print         62 48
                LD A.1             62 49
print:          CALL 5120          205 0 20
                RL D               203 18
                DJNZ loop          16 241
                RET                201
```

Figur .Program til konvertering fra decimal til binær.

De to koder der lagres i " A "-registret er ASCII-koderne for henholdsvis tallet 0 og 1.

Instruktionen " RL D " er gennemgået i afsnittet om " Rotationer og skifts". " RL D " foretager en rotation af samtlige bit i " D "-registret. Bit nr. 6 flyttes således til venstre over i bit nr. 7, bit nr. 7 flyttes til CARRY-flaget, CARRY til bit 0 osv. Dette foretages 8 gange hvorved samtlige bits er blevet testet og udskrevet på skærmen.

Lageradressen 32000 (= 0+256*125) er tilfældigt valgt og du kan derfor udskifte denne, hvis du har lyst. Nedenfor er vist et tilhørende BASIC-program som drager nytte af maskinkode programmet ved at POKE tal ned i adresse 32000, som derefter omsættes:

```
linienr.  kommando

    10    INPUT " indtast tal (0-255) ";a
    20    POKE 32000,a
    30    CALL start
```

hvor start angiver adressen du vælger at lægge programmet i.

Nu foretager " BIT " kun test af enkelt byte registre og lagerceller, men du kan naturligvis selv udvide programmet til at kunne teste i 16-bit tal, registre og lageradresser. Dette ville kræve at du foretager en rotation i begge bytes, .d.v.s. såvel i low- som highbyten.

2. SET - INSTRUKTIONEN:

Med denne instruktion kan enkeltbit 'kommanderes' til at blive "set". Igen kan hver enkelt register blive adresseret foruden den indirekte adressering, som tillader os også at ændre i alle lageradresser.

Instruktionen generelle format ser således ud:

SET 5.B

som vil gøre " B "-registrets bit nr. 5 eet, d.v.s. "set".

Der kan "set"-tes enkelt bit i ialt 7 registre samt lageradresser, som vist nedenfor:

Bit nr. / Register	0.	1.	2.	3.	4.	5.	6.	7.
A	199	207	215	223	231	239	247	255
B	192	200	208	216	224	232	240	248
C	193	201	209	217	225	233	241	249
D	194	202	210	218	226	234	242	250
E	195	203	211	219	227	235	243	251
H	196	204	212	220	228	236	244	252
L	197	205	213	221	229	237	245	253
(HL)	198	206	214	222	230	238	246	254
(IX+n)	198	206	214	222	230	238	246	254
(IY+n)	198	206	214	222	230	238	246	254

Tabel. Oversigt over "SET"-instruktionerne.

Igen skal de indekserede instruktioner foranstilles koderne:

IX: 221 203 n

IY: 253 203 n

syntes ligegyldig, hvilket imidlertid IKKE er tilfældet. I stedet for at bruge en hel byte som signal benytter man ofte enkelt bits hvorved man sparer meget plads. Dette er f.eks. tilfældet i AMSTRADs operativsystem, hvor markeringer af at bestemte ting er indtruffet lagres i enkelt bits. På samme måde kunne du anvende disse instruktioner til at signalere at du har gennemløbet et bestemt loop etc.

2. RESET - INSTRUKTIONEN:

Med denne instruktion kan enkeltbit 'kommanderes' til at blive "reset". Igen kan hver enkelt register blive adresseret foruden den indirekte adressering, som tillader os også at ændre i alle lageradresser.

Instruktionen generelle format ser således ud:

RES 5.B

som vil gøre " B "-registrets bit nr. 5 nul, d.v.s. "reset".

Der kan "reset"-tes enkelt bit i ialt 7 registre samt lageradresser, som vist nedenfor:

Bit nr. / Register	0.	1.	2.	3.	4.	5.	6.	7.
A	135	143	151	159	167	175	183	191
B	128	136	144	152	160	168	176	184
C	129	137	145	153	161	169	177	185
D	130	138	146	154	162	170	178	186
E	131	139	147	155	163	171	179	187
H	132	140	148	156	164	172	180	188
L	133	141	149	157	165	173	181	189
(HL)	134	142	150	158	166	174	182	190
(IX+n)	134	142	150	158	166	174	182	190
(IY+n)	134	142	150	158	166	174	182	190

Figur . Oversigt over " RES "-instruktionen.

Igen skal indeksinstruktionerne foranstilles de sædvanlige:

IX: 221 203 n
IY: 253 203 n

FLAGEFFEKT:

Hverken "SET" eller "RES"-instruktionerne påvirker ikke flagene i "F"-registret.

BLOK - LAGRINGS INSTRUKTIONER:

Vi har tidligere gennemgået instruktioner, der kunne lagre enkelt bytes i registre og lageradresser. I dette afsnit skal vi se på 4 instruktioner, som er sammensat af bl.a. enkelt byte lagrings instruktioner, men også af andre. De instruktioner som den sammensatte instruktion er sammensat af udføres automatisk blot ved udførelse af den sammensatte instruktion. Fordelen ved disse instruktioner er naturligvis både tids- og byte besparelser.

de fire instruktioner ses nedenfor i figuren længst mod venstre. Derefter kommer den sædvanlige decimal kode, som skal bruge til at POKE instruktionerne ned i AMSTRADs lager. Længst mod højre er vist hvilke instruktioner der ligger inde i de sammensatte:

Assembler	Decimal kode		Sammensat af

LDI	237 160		LD A,(HL)
			LD (DE),A
			INC DE
			INC HL
			DEC BC

LDID	237 168		LD A,(HL)
			LD (DE),A
			DEC DE
			DEC HL
			DEC BC

LDIR	237 176	loop:	LD A,(HL)
			LD (DE),A
			INC DE
			INC HL
			DJNZ loop

LDDR	237 184	loop:	LD A,(HL)
			LD (DE),A
			DEC DE
			DEC HL
			DJNZ loop

Figur . Oversigt over blok-lagrings instruktionerne.

Du kan se af de instruktioner, som blok lagringsinstruktionerne er sammensat af, fungerer meget ens. Forskellen ligger som med sammenligningsinstruktionerne kun i at de par arbejder nedad henholdvis opad i lageradresserne og at de to nederste er 'automatiske'.

Generelt kan det siges at " HL " lagres med en startadresse, hvis indhold flyttes til indholdet af adressen i " DE "-registerparret. Denne form for direkte lagring mellem adresser kræver med de normale instruktioner flere bytes, mens de sammensatte kan udføre dette med kun 2. Du skal imidlertid altid også tage den tid instruktionen er om at blive udført med i betragtning. Når overførslen er sket enten øges eller mindskes " DE " og " HL ", mens " BC " altid mindskes. Dette registerpar kan således ofte bruges som tæller.

De to første instruktioner kaldes de halvautomatiske, idet disse kun foretager flytning af een byte - ligegyldigt hvad der er lagret i tælleren. Således skal disse to instruktioner udføres netop det antal gange du ønsker at flytte bytes. De to sidste indeholder loop-faciliteten som vist i figuren. Overførslen mellem " HL " og " DE " sker automatisk indtil " BC "-registret har nået værdien nul. Dette registerpar mindskes med een for hver bytes der flyttes.

Vi kan med de nye instruktioner effektivisere vores blokprogrammer fra tidligere afsnit. Nedenfor er vist hvordan:

	Assembler	Decimal kode
	LD HL.(adr-1)	42 n n
	LD DE.(adr-2)	237 91 n n
	LD BC.(adr-3)	237 75 n n
	XOR A	175
	ADD A.B	128
	JR Z.lowbyte	40 10
bloop:	PUSH BC	197
	LD B.255	6 255
	LDIR	237 176
	RET C	216
	DJNZ bloop	16 247
	XOR A	175
	ADD A.C	129
	RET Z	200
	SCF	55
	LD B.A	71
	JR lloop	24 243

Figur . Effektiviseret blokflytningsprogram.

Flageffekten for denne gruppe instruktioner er vist i Appendix.

RESTART INSTRUKTIONERNE:

" RST " eller restart instruktionerne svarer nogenlunde til " CALL " instruktioner, som bare har fået fastlåst de adresser der kan kaldes. Der er således kun 8 adresser der kan kaldes med restartinstruktionerne. Disses generelle format er vist nedenfor, hvor:

```
10001   ...
10000   RST 8
```

der ligsom med " CALL " lagres adressen lige efter instruktionen og hoppes til den angivne adresse. her lagres således adresse 10001 på stakken og der hoppes til adresse 8.

Som du ser sparer disse instruktioner os for to bytes og er tilmed meget hurtige i forhold til CALL. Dette er grundet til at folkene bag AMSTRAD har udnyttet de 8 instruktioner til at styre AMSTRAD. Instruktionerne har således vigtige funktioner for AMSTRADs operativsystem.

De 8 instruktioner er vist nedenfor:

Assembler	Decimal kode
RST 0	199
RST 8	207
RST 16	215
RST 24	223
RST 32	231
RST 40	239
RST 48	247
RST 56	255

Figur . Oversigt over restart-instruktionerne.

Som nævnt svarer de 8 restartinstruktioner til " CALL "-instruktioner, der blot kaldes adresserne 0, 8, ..., 56. Det er derfor interessant at se hvad der ligger på disse adresser. Dette er vist på sidste side i afsnittet samt programmet der gør det muligt at aflæse adresserne.

Samtlige restart instruktioner i AMSTRAD er udformet til at skulle kalde rutiner i Operativsystemet, d.v.s. på adresserne 0 til 16384, i BASIC-fortolkeren, som ligger på adresserne 49152 og frem og endelig dine rutiner, som kun kan lagres i RAM.

Du kan måske undre dig over at BASIC-fortolkeren kan ligge på samme adresser som skærmbilledet. Dette hænger sammen med den måde hvorpå maskinen, d.v.s. hardware, er konstrueret på. Ved at sende et bestemt signal til mikroprocessoren vil denne begynde at læse i BASIC-fortolkeren - sendes herefter et andet signal til den vil den læse skærmbilledet.

" RST 0 "-instruktionen udfører en komplet sletning og klargøring af AMSTRAD svarende til det der sker når du tænder for maskinen. Du kan også få dette udført medens maskinen er tændt med BASIC-kommandoen:

```
CALL 0
```

hvilket vil kalde den kode der ligger på adresse 0 i RAM-en. Da denne er identisk med den der ligger i ROM-en bliver maskinen "reset", som man kalder det. De instruktioner der udføres ved denne kommandoer er:

Assembler	Decimal kode
LD BC.32649	1 137 127
OUT (C).C	237 73
JP 1408	195 128 5

De to første instruktioner sørger for at Operativsystemet på adresse 0 til 16384 bliver læst af mikroprocessoren. Hoppet til adresse 1408 er således den rutine, der foretager sletningen etc.

En anden måde at kalde denne 'reset' rutine er ved at trykke på CTRL SHIFT ESC -knapperne samtidig.

" RST 8 " giver os mulighed for at kalde rutiner i såvel RAM som begge ROM-er. Adressen vi ønsker at kalde skal angives på en lidt speciel måde. Lige efter restart instruktionen skal adressen komme som et 16-bit tal (naturligvis). Adressen skal imidlertid deles op i 3 dele, når den angives:

1. del:

bit nr. 0-13 : selve adressen, der skal kaldes

Da en adresse altid fylder 16 bit mangler der således 2 bit, idet vi med 14 bit kun kan angive adresser af størrelsen 0 til 16384. Hermed kan vi IKKE kalde hele maskinens lager. De to manglende bits bruges til at signalere hvor i AMSTRADs lager man ønsker at kalde en rutine.

2. del:

```
bit nr. 14 : 0 = operativsystem
             1 = underliggende RAM
```

3. del:

```
bit nr. 15 : 0 = BASIC-fortolkeren
             1 = underliggende RAM.
```

Anden og tredie del af adressen, hvorved vi ialt har 16 bit, er således signaler til mikroprocessoren om hvor vi ønsker at kalde en rutine. Ordet 'underliggende' skal forståes som at de adresser som operativsystem henholdsvis BASIC-fortolker ligger på OGSÅ er adresser for RAM-områder. Du må derfor vælge hvilke af områderne du vil have.

Lad os se på et eksempel. Du har allerede se hvordan man kan få opstartmeddelelsen frem på skærmen. Dette skete ved:

```
Assembler       Decimal kode

LD BC.32649     1 137 127
OUT (C).C       237 73
LD HL.1683      33 147 6
CALL 1771       205 235 6
RET             201
```

Dette er netop et kald af en rutine i AMSTRADs operativsystem og vi skulle derfor kunne bruge " RST 8 "-instruktionen i stedet. Adressen vi skal kalde er 1771 hvorved vi har de første 14 bit i den adresse der skal komme efter restartinstruktionen:

```
Assembler

RST 8
??00 0110 1110 1011
```

Tallet efter restart 8 er adresse 1771 omsat til binær skrivemåde. '?' antyder at vi mangler disse to bits.

Bit 14 skal enten være 1 eller 0. Sætter vi den 1 signalerer vi til AMSTRAD at vi ønsker at bruge RAM-området der ligger under operativsystemet. Vi kan herved IKKE kalde nogen rutiner i operativsystemet, idet AMSTRAD vil læse RAM-adressernes indhold og ikke ROM-ens. Da rutinen vi skal bruge NETOP ligger i operativsystemet skal bit 14 gøres 0:

```
Assembler

RST 8
?000 0110 1110 1011
```

Tilsvarende med bit 15. Gøres den 1 signalerer vi til AMSTRAD at vi ønsker at bruge RAM-området under BASIC-fortolkeren. Hermed er det slut for AMSTRAD at forstå BASIC - indtil vi har signaleret tilbage. Umiddelbart skal vi ikke bruge BASIC-fortolkeren i vores rutine så vi kan gøre bit 15 1:

```
Assembler

RST 8
1000 0110 1110 1011
```

Denne binære adresse svarer til 34539, og det vi har gjort ved at sætte bit 15 til 1 er blot at addere 32768 til vores adresse. Dette tal svarer netop til 2 opløftet til 15. Tilsvarende hvis vi skulle have bit 14 til 1 skulle vi addere 16384.

Programmet skulle således kunne ændres til:

```
Assembler       Decimal kode

LD HL.1683      33 147 6
RST 8           207
DW 34539        235 134    (= 1771+32768)
RET             201
```

Figur . Anvendelse af " RST 8 ".

Ordet " DW " benyttes for forkortelsen Define Word eller på dansk 16-bit tal definition. Dette angiver at vi mener noget bestemt med dette tal.

Når du har fået udført dette program skulle du prøve at bede AMSTRAD læse BASIC-fortolkeren i stedet for RAM-området derunder. Prøv at se hvad der sker på skærmen !. Hvorfor ? - find selv svaret.

Koden der udføres ved " RST 8 " er vist nedenfor:

Adresse	Assembler
00008	JP 47490
	JP 47484
	PUSH BC
	RET

Husk at de viste adresser skal tolkes som den du angav ovenfor, d.v.s. med bit 14 og 15 som signalangivere. Der ligger nemlig ingen maskinkode programmering på adresserne 47490 osv. Dette er jo DIT BASIC-RAM område.

" RST 16 " benyttes ikke i AMSTRAD standartsystemet, idet denne restart instruktion skal bruges til at kalde ROM-udvidelser, der kan kobles bag på AMSTRAD. Når instruktionen udføres hopper AMSTRAD som vist:

Adresse	Assembler
00016	JP 47638
	JP 47632
	PUSH DE
	RET

Igen skal adresserne tolkes med hensyn til bit 14 og 15, som ovenfor. Dette gælder samtlige adresser i forbindelse med RST.

" RST 24 " bruges som " RST 8 " til kald af rutiner i AMSTRADs ROM eller RAM, men begrænser sig ikke kun til de først og sidste 16 Kilobytes, som det er tilfældet med " RST 8 ". Dette er opnået ved at bruge endnu en byte til angivelse af den adresse man ønsker at kalde. Således skal " RST 24 " efterfølges af en adresse PLUS en ekstra byte - d.v.s. ialt 3 bytes skal komme efter instruktionen. Disse tre bytes kan imidlertid placeres overalt i AMSTRADs lager. Hvordan finder " RST " instruktionen dem så ?. Jo, umiddelbart efter restartkommandoen angives adressern på stedet hvor de tre bytes er placeret. Skal vi eksempelvis kalde en rutine som ovenfor i operativsystemet kunne dette se således ud:

```
Adresse         Assembler

10000           RST 24
10001           DW 20000
...

...
20000           DW 1771
                DB  253
```

Figur . Kald af rutine i Operativsystemet med " RST 24 ".

Denne form for kald kaldes parameterblok adressering. De tre bytes er parameterblokken, og adressen efter restartrutinen er adressen på parameterblokken.

Idet selve adressen, d.v.s. de to første byte ikke skulle volde problemer skal vi se på den tredie byte der følger efter restart instruktionen. Som enhver anden byte kan denne antage værdier fra 0 til 255 (regnet uden fortegn). Fire af disse værdier er fastlagt til brug i AMSTRAD. De øvrige kan frit benyttes til bl.a. kald af tilkoblede ROM-er i form af spil etc.

De fire 'faste' koder er og betyder:

Kode	Nedre 32 K	Øvre 32 K
252	Operativsystem	BASIC-fortolker
253	Operativsystem	RAM-område
254	RAM-område	BASIC-fortolker
255	RAM-område	RAM-område

og hermed lever AMSTRAD folkene op til deres løfte om at der rent faktisk kan benyttes fulde 64 Kilobyte til brugerprogrammering. Al ROM er herved koblet ud og du har en helt 'ren' maskine - EJ AT GLEMME INGEN BASIC INTET OPERATIVSYSTEM TIL AT HJÆLPE DIG, hvis du laver fejl. Der er kun et crash tilbage.

Du skulle hermed have forstået eksemplet vist ovenfor, hvor den tredie byte i parameterblokken var 253. Dette svarer jo til eksemplet i forbindelse med " RST 8 ". Kode 253 betyder jo at vi i de første 32 Kilobyte (nederste) har Operativsystemet liggende, nemlig i form af en ROM, og i de sidste 32 Kilobyte har brugerområde til vores programmer.

Koden der ligger på adresse 24 er vist nedenfor:

Adresse	Assembler
00024	JP 47551
	JP 47537
	JP (HL)
	NOP

" RST 32 " kan bruges til at aflæse indholdet i RAM-områder overalt i AMSTRAD. Dette sker uafhængigt af de ovenfor nævnte tilstande, d.v.s. ligegyldigt hvad du har valgt at AMSTRAD skal læse på de forskellige adresser, kan du med restart 32 altid læse indholdet i RAM-en. Du kan således gøre brug af mere end 64 Kilobyte med visse modifikation.

RAM-adressen du ønsker at læse i skal lagres i " HL "-registerparret, idet aflæsningen sker vis en " LD A.(HL) " instruktion. Eksempelvis kan vi læse RAM-området fra 49152 og 255 bytes frem med:

	Assembler	Decimal kode
	LD HL.49152	33 0 192
	LD B.255	6 255
	LD DE.20000	17 32 78
loop:	PUSH BC	197
	RST 32	231
	LD (DE).A	18
	INC HL	35
	INC DE	19
	POP BC	193
	DJNZ loop	16 248
	RET	201

Figur . Kopiering af RAM-område.

Koden der udføres med en " RST 32 " er vist nedenfor:

Adresse	Assembler
00032	JP 47819
	JP 47545

" RST 40 " benyttes til at kalde en rutine i AMSTRADs operativsystem svarende til " RST 8 " blot UDEN bit 14 og 15 signalerne. Dette skyldes at operativsystemet jo kun fylder de første 16 Kilobyte. Adressen i operativsystemet skal følge efter restart instruktionen umiddelbart.

Koden der udføres med en " RST 40 " instruktion er blot et simpelt hop:

Adresse	Assembler
00040	JP 47662

" RST 48 " benyttes ikke af AMSTRAD-systemet.

" RST 56 " er del af Z-80 mikroprocessorens interrupt system, som vi endnu ikke har talt om. Vi skal i et senere afsnit kort omtale dette og hvordan vi kan udnytte det. Her skal blot kort siges følgende:

AMSTRAD kan være i eet af 3 insterrupt mode's, hvor 'mode' betyder (lidt simpelt) humør. Et interrupt er et signal, der beder mikroprocessoren om at standse sit travle arbejde og foretage noget andet - indtil den igen får lov til at vende tilbage til sit normale arbejde. Afhængig af hvilket humør (mode) at AMSTRAD er i vil den behandle afbrydelserne. Dens normale mode er nr. 1, hvilket netop er det mode som " RST 56 " hører til. Hver gang AMSTRAD bliver afbrudt (og er i mode 1) bliver en " RST 56 " instruktion udført. Koden der bliver udført ved restart 56 er:

```
Adresse         Assembler

00056           JP 47417
                RET
```

hvor returneringen netop er meldingen om at AMSTRAD kan fortsætte sit normale arbejde. Adressen der hoppes til, hvilket jo er

```
47417 - 32768 = 14649
```

er starten på AMSTRADs fantastiske interrupt behandlingssystem. Dette sørger som bekendt for de interruptfaciliteter vi har i BASIC, bl.a. ON BREAK GOTO, ON ERR .., DI, EI osv..., ej at glemme lydfaciliteterne m.m. Med andre ord, programmet der starter på ovennævnte adresse i operativsystemet er yderst interessant.

Den måske aller mest spændende ting ved disse restartfunktioner er at vi kan bruge dem i vor egen programmering. Dette er ikke muligt i lignende Z-80 baserede computere. Eksempelvis kunne vi bruge " RST 0 " til at kalde vores eget program, som vist nedenfor:

```
Adresse         Assembler

00000           JP 42000
...

...
20000           RST 0
20001           RET
...

...
42000           LD HL.1683
42003           CALL 1771
42006           RET
```

Figur . Program til udnyttelse af " RST " faciliteterne.

BASIC-programmet kunne se således ud:

```
Linienr.  Kommando

    10    REM **** RST 0 kald ******
    20    FOR a=1 TO 7
    30    READ f
    40    POKE a+41999,f
    50    NEXT a
    60    DATA 33,147,6,205,235,6,201
    70    POKE 0,195
    80    POKE 1,16
    90    POKE 2,164
   100    POKE 20000,199
   110    POKE 20001,201
```

Linienummer 70 til 90 lagrer kaldet til adresse 42000 i RAM-adresserne 0 til 2. Linie 100 og 110 indeholder restartinstruktionen samt instruktionen til returnering til BASIC. Prøv programmet - selve funktionen må være gammel kendt nu.

Det ovenfor viste kan lade sig gøre med samtlige " RST "-instruktioner men KUN sålænge AMSTRAD kan læse bruger RAM-en under operativsystemet.

INTERRUPT - INSTRUKTIONERNE:

Interrupt betyder 'at afbryde' og det er lige havd instruktionerne i denne gruppe bruges til. For at AMSTRAD skal kunne opfatte dine indtastniger m.m. må vi sørge for at den lytter på de kanaler vi har til at snakke til den med. Ellers ville den blot stå og hygge sig og lave ingenting. Enhver form for snak mellem AMSTRAD og omverdenen, om det så er dine indtastninger, signaler til og fra cassetten, diskdrive (forhåbentlig snart), skærmen eller andet, så kaldes dette for INPUT/OUTPUT eller I/O. Ved konstant at afbryde AMSTRAD og få den til at lytte til vores indtastniger eller til cassetten kan vi komme i kontakt med maskinen - og dette er netop hvad interruptsystemet bruges til (bl.a.). Da der kan forekomme mange afbrydelser, f.eks. at du konstant tramper på tasterne eller at casetten konstant ønsker at sende data, må interruptsystemet være hurtigt for at kunne give AMSTRAD besked om alt hvad der sker. Det er systemet bestemt også. 50 gange i sekundet bliver AMSTRAD afbrudt i sit arbejde og må ud og lytte. Er der imidlertid ikke noget der kommer ind til den, kan den straks efter genoptage sit sædvanlige arbejde. Dette sker ligegyldigt hvad AMSTRAD laver, d.v.s. såvel når den udfører BASIC- som maskinkodeprogrammer, eller når den ikke laver noget. Imidlertid kan systemet med insterruptene slås fra, så AMSTRAD kan få arbejdsro - og så går det endnu stærkere med at få lavet noget.

Inde i AMSTRAD sidder en lille men meget præcis klokke (et kvarts krystal) som sørger for at AMSTRAD bliver afbrudt med regelmæssige tidsrum. Denne klokke medvirker også til at holde den klokke vi kan bruge fra BASIC i gang.

Det at AMSTRAD skal ud og lytte til indtastninger kaldes for scanning af tastaturet. Dette sker een gang for hver interrupt, d.v.s. også 50 gange i sekundet. AMSTRAD opsamler (næsten) alle dine indtastninger i et lager, en såkaldt buffer. Her ligger indtastningerne indtil der er tid til at gøre noget ved indtastningerne. Er det et program der skal køre lagres de linier der skal udføres i bufferen.

Nedenfor er vist en instruktion som kan slå interruptsystemet fra hvorved AMSTRAD ikke vil kunne 'høre' hvad du indtaster:

DI

hvilket står for Disable Interrupt, eller stop interruptsystem. Du skal immidlertid være varsom med at slå dette system fra, hvis du ikke senere i dine programmer sætter det til igen.

Alt det ovenfor nævnte sker når AMSTRAD er i et af tre mulige modes, nemlig mode 1. Du kan i princippet bestemme hvilket mode at AMSTRAD skal være i med en af instruktionerne i denne gruppe. Ønsker du at AMSTRAD skal være i mode 1 udfører du blot:

```
IM 1
```

forkortet for Interrupt Mode nr. 1. Hermed opfører AMSTRAD sig som du er vant til. Der findes imidlertid to andre modes, nemlig 0 og 2.

Mode 1 tillader en ydre tilkoblet enhed, f.eks. en printer, en anden computer el.a. at snakke og forstyrre AMSTRAD. Dette gøres ved at den ydre enhed aktiverer en kontrollinie ind til AMSTRADs mikroprocessor. Denne stopper herved øjeblikkelig sit sædvanlige arbejde og begynder at lytte til hvad den ydre enhed fortæller.

Det sidste mode, mode 3, kan bruges af dig til at få AMSTRAD til at udføre periodiske opgaver, d.v.s. opgaver som skal ske igen og igen med et givet tidsrum imellem. Sætter du AMSTRAD i mode 3 ophører scanningen af tastaturet osv., men det behøver ikke være noget problem idet du selv kan sørge for at AMSTRAD kalder de rutiner der udfører scanningen.

Når AMSTRAD bliver afbrudt af et interrupt (og er i mode 3) vil mikroprocessoren modtage information om hvor den skal fortsætte, via del databussen dels " I "-registret.

Vi har ikke talt hardware overhovedet i denne bog, men kort fortalt modtager mikroprocessoren alle data via databussen, som er 8 elektriske ledninger, hvorigennem de 8 bit en byte består at kommer. Da en bit kan være enten "set" eller "reset" signalerer strømstyrken i ledningerne om bitten skal være det ene eller det andet. Foruden databussen har AMSTRAD også en adressebus der består af 16 elektriske ledninger, som tilsvarende kan angive 16 bits værdi. Med disse 16-bit kan en adresse udpeges, og vi kan med denne bus aflæse eller skrive i enhver adresse i AMSTRADs lager (dog kun i RAM).

Som nævnt giver databussen et 8-bit tal til mikroprocessoren. Disse 8 bit er lowbyten i adressen hvorfra der skal fortsættes udførelsen af instruktioner indtil afbrydelsen er færdig eller indtil næste afbrydelse kommer. Highbyten i adressen får AMSTRAD som nævnt fra " I "-registret, som netop hedder I på grund af tilknytningen til Interruptsystemet. Egentlig skal AMSTRAD ikke fortsætte på den angivne adresse. Den skal faktisk kun betragte den angivne (og den efterfølgende) adresses indhold hvorved den rigtige adresse fremkommer. Forestil dig f.eks. at databussen og " I "-registret tilsammen indeholder adressen 20000. Indholdet i de 20000 OG 20000+1 aflæses så og opfattes som en ny adresse. Står der f.eks. 100 i 20000 og 1 i 20001 bliver den nye adresse 100+1*256 = 356. På denne adresse fortsætter mikroprocessoren så udførelsen. Den aflæser de instruktioner der måtte stå i 356 og udfører dem. Dette bliver den ved med indtil den bliver bedt om at returnere fra interruptrutinen ELLER at et nyt interrupt afbryder den igen.

Det er muligvis en smule indviklet, men jeg håber du vil forstå princippet - da det giver os nogle meget stærke programmeringsfaciliteter. Sidst i afsnittet skal vi se på et eksempel.

Instruktionerne i denne gruppe er vist nedenfor:

Assembler	Decimal kode
EI	251
DI	243
IM 0	237 70
IM 1	237 86
IM 2	237 94
RETI	237 77
RETN	237 69

Figur . Oversigt over Interruptinstruktionerne.

De to første instruktioner henholdsvis tænder og slukker interruptsystem. " EI " er forkortet for Enable Interrupt, mens " DI " er forkortet for Disable Interrupt. Du skal ALTID huske at tænde interruptsystemt igen, hvis du ønsker at AMSTRAD skal kunne opfatte dine indtastninger.

De tre næste instruktioner kan bruges til at sætte AMSTRAD i et bestemt Interrupt Mode. Vi har ovenfor set på de tre modes. Også her er det vigtigt at stille AMSTRAD tilbage I Interrupt Mode 1 hvis du eventuelt ændrer det.

Endelig er der de to returnerings-instruktioner. Den første er forkortet for Return from Interrupt eller returner fra interruptrutinen. Det er denne instruktion, der skal udføres, når AMSTRAD skal tilbage og genoptage sit sædvanlige arbejde, når maskinen er i en interruptrutine.

Hvad er en interruptrutine ???. Det er faktisk ikke noget specielt, men man siger at AMSTRAD udfører en interrupt rutine, når et program bliver udført som følge af et interrupt. F.eks. det at scanne tastaturet for indtastninger kunne kaldes en interruptrutine, idet denne rutine kaldes hver gang der forekommer et interrupt. Du kan imidlertid også lave interruptrutiner - dette bliver vist nedenfor.

" RETI " fungerer således ligsom " RET ", ved at de to 'sidste' (nederste) bytes tages af stakken og lagres i Program Tælleren. Denne begynder så udførelsen af programmer fra denne adresse. Forskellen mellem " RET " og " RETI " er at man kan koble hardware udstyr på mikroprocessoren, som kan opfatte udførelsen af " RETI "-instruktionen. Denne er jo et signal til mikroprocessoren om at interruptrutinen afsluttes og at det afbrudte program kan genoptages. Hardware der kan kobles på kan 'føle' dette og udføre funktioner der udbygger mikroprocessorens interruptsystem. Dette anvendes ofte i større datamater, som har prioriteret interruptsignalerne så nogle er mere vigtige end andre. Adskillelsen mellem disse kan styres af det tilkoblede hardware (andre mikroprocessorer o.l.).

Internt i AMSTRADs mikroprocessor findes imidlertid også en slags prioritering af interruptsignaler. Således findes de anmindelige signaler og endelig de der kaldes 'non-maskabel'. De sidste kan man ikke generere fra et program, men det kan lade sig gøre fra en anden hardware enhed koblet på bagsiden af AMSTRAD. denne type interrupt kan IKKE afbrudes af et nyt interrupt - i modsætning til almindelige interrupts. Dette betyder at hvis der opstår et nonmaskable interrupt vil dette standse mikroprocessorens arbejde. Og ikke før en " RETN " instruktion er udført vil AMSTRAD kunne fortsætte. N'et tilsidst antyder netop navnet på interrupttypen Non-mask'able.

Ovenfor blev det nævnt at et almindeligt interrupt kunne afbrydes af et nyt almindeligt interrupt. Dette gælder imidlertid ikke hvis det første interrupt har slået interruptsystemet fra. Dette svarer til at du sidder i stuen og lytter til musik. Pludselig ringer telefonen og du må op og tage den. Tager du derimod hovedtelefoner på mens du hører musik kan telefonen ringe nok så meget - du vil ikke høre den - du har slået interruptsystemet fra.

Vi skal nu til at se på hvordan du kan udnytte interruptsystemet til egen brug. Vi skal altså have en rutine, der skal udføres med et fast tidsrum imellem. Det kunne f.eks. være et ur eller lignende. Jeg vil lade dig programmere et selv. Vi skal så angive over for AMSTRAD at den ved hvert interrupt skal springe til adressen, hvor dit program står og udføre dette. Hertil skulle vi bruge " I "-registret samt databussen. Databussen indeholder normalt lutter aktiverede linier, hvilket svarer til at samtlige bits er "set". Databussens bidrag til adressen er således tallet 255. Vi skal så lagre highbyte delen af adressen i " I "-registret. Vi har herved mulighed for at pege på adresser fra 255 og et multiplum af 256, idet " I "-reistret jo er highbyten. Lagrer vi f.eks. tallet 20 i " I "-registret vil AMSTRAD aflæse adresse 255+20*256 = 5375 og 5376. Indholdet i disse to adresser bliver opfattet som en ny adresse, hvorfra AMSTRAD vil hente de næste instruktioner der skal udføres. Ved at lagre " I "-registret med en værdi og placere adressen på dit program i adressen som udpeges via databussen og " I "-registret har du lavet din egen interruptrutine. Du kan herved få det til at se ud som om AMSTRAD både udfører dit BASIC-program samtidig med din interruptrutine.

Instruktionerne hertil er vist nedenfor:

Start af interruptrutinen:

Assembler	Decimal kode
LD A.tal	62 n
LD I.A	237 71
IM 2	237 94

AMSTRAD er nu i Interrupt Mode 2 og vil ved næste interrupt tage værdien fra " I "-registret og databussen og finde først adressen, dernæst interruptrutinen. Denne bør du indlede således:

Interruptrutine :

Assembler	Decimal kode
DI	243
PUSH BC	197
PUSH DE	213
PUSH AF	245
PUSH HL	229
CALL adresse	
POP HL	225
POP AF	241
POP DE	209
POP BC	193
EI	251
RET	201

Som du ser lagrer rutinen blot indholdet i registrene på stakken og lægger dem tilbage efter at rutinen er færdig. Skulle AMSTRAD være afbrudt med vigtig information i registrene er det MEGET vigtigt at denne information ligger korrekt, når der returneres fra interruptet. Bemærk at aftryksregistrene ikke 'sikkerheds-kopieres', hvilket de egentlig burde.

Når du ikke længere ønsker at holde din interruptrutine i gang kan du slukke den med:

Assembler	Decimal kode
LD A.62	62 62
LD I.A	237 71
IM 1	237 86

som stiller AMSTRAD tilbage i det normale interrupt mode.

Når du laver din interruptrutine behøver denne naturligvis ikke standse interruptsystemet, som det er vist ovenfor. Du skal så bare være opmærksom på at hvis din interruptrutine tager mere end 0.02 sekund at udføre, så vil et nyt interrupt fremkomme og din rutine vil straks starte forfra - uden at blive færdig - nogensinde !.

Hvad kunne man nu forestille sig at bruge interruptfaciliteterne til ?. Jo du kender vel allerede de fantastiske spil man kan få til maskinen. Foruden animation, bevægelse og farver giver spillene også lyde som eksplosioner etc. Disse faciliteter er mulige p.g.a. interruptfaciliteterne. Da AMSTRAD afbrydes 50 gange PR. SEKUND. mærker du ikke den lille forsinkelse der fremkommer ved at en tone skal holdes i gang, SAMTIDIG med at et dyr el.lign. skal bevæge sig på skærmen.

Du skal dog advares imod at blive alt for skuffet over dine egne anstrengelser over for simple programmer. De fleste (samtlige) spil af blot rimelig kvalitet udvikles på STØRRE computere med brug af specielle sprog, som gør skærmbehandling lettere (end fra maskinkode). De store computere kan så når programmet er færdigt lave programmet om til maskinkode som AMSTRAD kan udføre. Man sidder derfor ikke og udvikler spil på en AMSTRAD - selvom det er på denne maskine de skal køre.

Som nævnt kunne dette kun ske ved at bringe AMSTRAD i interrupt mode 2.

Dette gør vi derfor først. Samtidig skal vi sørge for at AMSTRAD modtager adressen, hvor rutinen befinder sig, når første interrupt indfinder sig.

Forestil dig f.eks. at vi vælger at lægge adressen på den rutine, der skal være interruptrutine i adresse 16127.

IN / OUT - INSTRUKTIONERNE :

Disse instruktioner svarer til kommandoerne i BASIC med samme navn. Instruktionerne generelle format er således:

IN register.(port)

og

OUT (port).register

hvor registeret er et af 8-bit registrene. Udtrykket 'port' skal ikke forvirre dig. Enport har et nummer knyttet til sig svarende til en adresse. Dette nummer benyttes kun til at adressere porten, når der f.eks. skal sendes noget gennem den eller modtages noget fra den. AMSTRADs mikroprocessor befinder sig på den ene side af portene, mens der på den anden kan kobles alt på. Eksempelvis aflæses tastaturet gennem porte.

Modtagelse af data via porten sker med " IN "-instruktionen, som er vist nedenfor:

Assembler	Decimal kode
IN A.(n)	219 n
IN A.(C)	237 120
IN B.(C)	237 64
IN C.(C)	237 72
IN D.(C)	237 80
IN E.(C)	237 88
IN H.(C)	237 96
IN L.(C)	237 104
INI	237 162
INIR	237 178
IND	237 170
INDR	237 186

Figur .Oversigt over " IN "-instruktionerne.

" BC "-registret bruges til at angive portnummeret for de instruktioner der er opbygget som:

IN register.(C)

og med læsning fra port 'n' angives adressen med 'n' og " A "-registret.

Som med de øvrige blokinstruktioner findes i denne gruppe to automatiske og to halvautomatiske. Instruktionerne er sammen sat af følgende:

```
IND:      (C)  ->  (HL)
          DEC B
          DEC HL
```

Instruktionen er forkortet for Input med decrement.

```
INDR:  loop:  (C) ->   (HL)
              DEC B
              DEC HL
              DJNZ loop
```

Instruktionen er forkortet for Input med decrement og repeat. 'Repeat' betyder -gentagelse-, nemlig indtil " B "-registret er nået nul.

```
INI:      (C)  ->  (HL)
          DEC B
          INC HL
```

Instruktionen er forkortet for Input med increment.

```
INIR:  loop:  (C) ->   (HL)
              DEC B
              INC HL
              DJNZ loop
```

Instruktionen er forkortet for Input med increment og repeat. 'Repeat' betyder -gentagelse-, nemlig indtil " B "-registret er nået nul.

Afsendelse af data via porten sker med " OUT "-instruktionen, som er vist nedenfor:

Assembler	Decimal kode
OUT A.(n)	211 n
OUT A.(C)	237 121
OUT B.(C)	237 65
OUT C.(C)	237 73
OUT D.(C)	237 81
OUT E.(C)	237 89
OUT H.(C)	237 97
OUT L.(C)	237 105
OUTI	237 163
OUTIR	237 179
OUTD	237 171
OUTDR	237 187

Figur .Oversigt over " OUT "-instruktionerne.

```
OUTD:     (C)  ->  (HL)
          DEC B
          DEC HL
```

Instruktionen er forkortet for Output med decrement.

```
OUTDR:  loop: (C) ->   (HL)
              DEC B
              DEC HL
              DJNZ loop
```

Instruktionen er forkortet for Output med decrement og repeat. 'Repeat' betyder -gentagelse-, nemlig indtil " B "-registret er nået nul.

```
OUTI:     (C)  ->  (HL)
          DEC B
          INC HL
```

Instruktionen er forkortet for Output med increment.

```
OUTIR:  loop: (C) ->   (HL)
              DEC B
              INC HL
              DJNZ loop
```

Instruktionen er forkortet for Output med increment og repeat. 'Repeat' betyder -gentagelse-, nemlig indtil " B "-registret er nået nul.

REST GRUPPEN:

Dette afsnit afslutter gennemgangen af instruktionerne med 6 instruktioner, som kun for fuldstændighedens skyld er medtaget. Disse 6 er vist nedenfor:

Assembler	Decimal kode
CCF	55
SCF	63
HALT	118
NOP	0
DAA	39
NEG	237 68
CPL	47

Figur . Oversigt over restgruppen af instruktioner.

De to første instruktioner styrer CARRY-flaget direkte. Dette gøres "set" med S(et) C(arry) F(lag). Hermed fik du også forkortelsen. Modsvarende gør C(lear) C(arry) F(lag) CARRYflaget "reset".

De to næste instruktioner kan få AMSTRADs mikroprocessor over i en venteposition. " HALT "-instruktionen sætter mikroprocessoren til at vente til næste interrupt. " NOP " får mikroen til at vente 2 maskincycler.

" DAA " er forkortet for Decimal Adjust A-register, d.v.s. tilpas resultatet til Decimal notation. Instruktionen benyttes hvis man bruger BCD-notation til at lagre tal i AMSTRAD. BCD står for Binær Codet Decimal, og er således en anden måde at skrive tal på end ren binær. Med BCD skrivemåden 'pakker' man tal tættere sammen i hver byte. En byte er jo på 8 bit, og ved kun at bruge tallene 0 til 9 kan man skrive tal af størrelsen 0 til 99 i en byte. Dette medfører ofte en bedre præcision, når der skal regnes med større tal.

De 4 bits forskellige kombinationer er vist nedenfor:

Decimal værdi	BCD
00	0000
01	0001
02	0010
03	0011
04	0100
05	0101
06	0110
07	0111
08	1000
09	1001

Fra 10 og op til 16 er således ikke tilladte koder i BCD. Forestil dig som et eksempel at vi har lagret værdien 39 i een byte og 22 i en anden. Summen skal herefter beregnet:

```
BCD kode for 39:    0011 1001
 -   --   - 22:    0010 0010

Summen          :   0101 1011
```

Skal vi nu tolke summen som et BCD-tal får vi problemer idet tallet i de 4 laveste (nibbles) ikke er defineret. Vi kan således IKKE blot skrive vores maskinkodeprogram som:

```
Assembler       Decimal kode

LD A.39 bcd     62 '39'
ADD 22  bcd     198 '22'
```

idet vi får ikke det korrekte resultat. Problemet løses ved at summere tallet 6 til de 4 laveste bits. Gør vi det får vi tallet:

```
Summen          :   0101 1011
Plus 6          :   0000 0110

Korrekt BCD sum:    0110 0001
```

idet BCD-summen nu er de korrekte 61. Vores additionsprogram kan også få dette resultat, hvis vi blot tilføjer instruktionen " DAA ", som sørger for at tilpasse resultatet til BCD-koden. Der er mange flere faciliteter gemt i denne instruktion, men BCD-koden og regnereglerne skal først gennemgåes fra bunden af. Søg derfor til en bog herom.

Endelig har vi to logiske instruktioner som bruges til at negere og komplementere indholdet i " A "-registret. At komplementere betyder at vende samtlige "set" bits til "reset" og omvent.

NYTTIGE ADRESSER:

På de næste par sider får du nogle ofte nyttige adresser du kan bruge i din maskinkode programmering. Enkelte også i BASIC-programmeringen. Det drejer sig om:

1. Udskrift af koden i " A "-registret på Monitoren.
2. Udskrift af tekst lagret i AMSTRADs RAM.
3. Styring af placeringen af tekst på Monitoren.
4. Monitor mode skift.
5. Start/stop af cassettemotor.
6. Pause til en tast trykkes.

Adresserne er fundet ved at disassemblere AMSTRADs BASIC-fortolker. Jeg vil opfordre dig til på egen hånd at fortsætte denne disassemblering, idet flere bl.a. Monitorstyrings rutiner ville være rare at kunne bruge direkte fra maskinkode.

1. Udskrift af koden i " A "-registret på Monitoren.

Vi har allerede set anvendelse af denne rutine et par gange, men den gentages her af hensyn til opslag. Rutinen ligger i AMSTRADs operativsystem på adresse 5120. Den kaldes med " A "-registret indeholdende ASCII-koden for det symbol, der skal udskrives på Monitoren.

Bemærk at denne rutine er temmelig uhensigtsmæssig at bruge i forbindelse med udskrift af flercifrede tal.

Eksempel på udprintning:

Assembler	Decimal kode
LD BC.32649	1 137 127
OUT (C).C	237 73
LD A.'1'	62 49
CALL 5120	205 0 20
RET	201

2. Udskrift af tekst lagret i AMSTRADs RAM.

Med denne rutine kan vi udprinte tekst lagret i AMSTRADs lager. Teksten skal være lagret i ASCII-koder og skal afsluttes med kode 0. Hele spektret af ASCII-koder kan bruges inklusive linieskift (Linefeed), Sideskift (Formfeed) m.m.

Rutinen befinder sig i AMSTRADs operativsystem på adressen 1771. Nedenfor er vist hvordan man kunne bruge rutinen:

Linienr. Kommando

```
05   D$=STR$(FRE(""))
06   FOR a=1 TO LEN(D$)
07   T(a)=ASC(RIGHT$(D$,a))
08   NEXT a
10   FOR a=1 TO 30
20   READ f
30   POKE a+41999,f
40   NEXT a
50   DATA 65,77,83,84,82,65,68,32,70,82,
     73,32,82,65,77,32,T1,T2,T3,T4,T5,0
60   DATA 33,16,164,223,235,6,252,201
70   CALL 42022
```

3. Styring af placeringen af tekst på Monitoren.

Med denne rutine kan du svarende til BASIC-kommandoen LOCATE placere tekst på Monitoren hvor det passer dig. Linie og Kollonne angivelsen sker gennem " HL "-registret, hvor " H "-registret skal indeholde kollonne nummer inden kaldet af rutinen, og " L " skal indeholde linienummeret.

Rutinen ligger i AMSTRADs BASIC-fortolker på adresse 47989, og skal kaldes med BASIC-fortolkeren 'paget' ind. Pagingen er omtalt ved bl.a. Restart instruktionerne.

4. Monitor mode skift.

Med denne rutine kan du skifte mode, svarende til BASIC-kommandoerne MODE n, hvor 'n' kan antage værdierne 0 til 2. Det mode du ønsker skiftet til lagres i " A "-registret forud for kald af rutinen.

Rutinen ligger i AMSTRADs BASIC-fortolker på adresse 48142 og skal som ovenfor være forberedt ved at BASIC-fortolkeren skal være paget ind forud for kaldet af rutinen.

5. Start/stop af cassettemotor.

Denne rutine giver dig mulighed for at starte og stoppe cassettebåndoptagerens motor, der trækker båndet forbi tonehovedet. Bemærk at selvom du skal loade et program ind i lageret så skal cassettemotor startes ved denne rutine. Dette sker ikke automatisk. Rutinen kan også kaldes fra BASIC.

Start af cassettebånd motoren sker ved kald af rutinen på adresse 48238. Herved startes motor direkte. Standsning af motoren sker ved kald af samme rutine blot 3 bytes længere oppe i lageret - adresse 48241.

6. Pause til en tast trykkes.

Denne rutine kan bruges til at indføje en pause indtil en tast trykkes - ofte anvendt i forbindelse med valgmuligheder ved MENUer etc. Rutinen bruges bl.a. ved loadning fra cassettebåndoptageren.

Du skal kalde rutinen på adresse 47878, som vil gå i et loop indtil der er trykket på en tast. BREAK-tasten fungerer som normalt. Aflæsning af " A "-registret ved returnering vil give dig koden for ned trykkede tast.

ASSEMBLER KODE SORTERET ALFABETISK

ADC A,A	143
ADC A,B	136
ADC A,C	137
ADC A,D	138
ADC A,E	139
ADC A,H	140
ADC A,L	141
ADC A,n	206 n
ADC A,(HL)	142
ADC A,(IX+n)	221 142 n
ADC A,(IY+n)	253 142 n
ADC HL,BC	237 74
ADC HL,DE	237 90
ADC HL,HL	237 106
ADC HL,SP	237 122
ADD A,A	135
ADD A,B	128
ADD A,C	129
ADD A,D	130
ADD A,E	131
ADD A,H	132
ADD A,L	133
ADD A,n	198 n
ADD A,(HL)	134
ADD A,(IX+n)	221 134 n
ADD A,(IY+n)	253 134 n
ADD HL,BC	9
ADD HL,DE	25
ADD HL,HL	41
ADD HL,SP	57
ADD IX,BC	221 9
ADD IX,DE	221 25
ADD IX,IX	221 41
ADD IX,SP	221 57
ADD IY,BC	253 9
ADD IY,DE	253 25
ADD IY,IY	253 41
ADD IY,SP	253 57
AND A	167
AND B	160
AND C	161
AND D	162
AND E	163
AND H	164
AND L	165
AND n	230 n
AND (HL)	166
AND (IX+n)	221 166
AND (IY+n)	253 166
BIT 0,A	203 71
BIT 0,B	203 64
BIT 0,C	203 65
BIT 0,D	203 66
BIT 0,E	203 67
BIT 0,H	203 68
BIT 0,L	203 69
BIT 0,(HL)	203 70
BIT 0,(IX+n)	221 203 n 70
BIT 0,(IY+n)	253 203 n 70
BIT 1,A	203 79
BIT 1,B	203 72
BIT 1,C	203 73
BIT 1,D	203 74
BIT 1,E	203 75
BIT 1,H	203 76
BIT 1,L	203 77
BIT 1,(HL)	203 78
BIT 1,(IX+n)	221 203 n 78
BIT 1,(IY+n)	253 203 n 78
BIT 2,A	203 87
BIT 2,B	203 80
BIT 2,C	203 81
BIT 2,D	203 82
BIT 2,E	203 83
BIT 2,H	203 84
BIT 2,L	203 85
BIT 2,(HL)	203 86
BIT 2,(IX+n)	221 203 n 86
BIT 2,(IY+n)	253 203 n 86

BIT 3,A	203 95
BIT 3,B	203 88
BIT 3,C	203 89
BIT 3,D	203 90
BIT 3,E	203 91
BIT 3,H	203 92
BIT 3,L	203 93
BIT 3,(HL)	203 94
BIT 3,(IX+n)	221 203 n 94
BIT 3,(IY+n)	253 203 n 94
BIT 4,A	203 103
BIT 4,B	203 96
BIT 4,C	203 97
BIT 4,D	203 98
BIT 4,E	203 99
BIT 4,H	203 100
BIT 4,L	203 101
BIT 4,(HL)	203 102
BIT 4,(IX+n)	221 203 n 102
BIT 4,(IY+n)	253 203 n 102
BIT 5,A	203 111
BIT 5,B	203 104
BIT 5,C	203 105
BIT 5,D	203 106
BIT 5,E	203 107
BIT 5,H	203 108
BIT 5,L	203 109
BIT 5,(HL)	203 110
BIT 5,(IX+n)	221 203 n 110
BIT 5,(IY+n)	253 203 n 110
BIT 6,A	203 119
BIT 6,B	203 112
BIT 6,C	203 113
BIT 6,D	203 114
BIT 6,E	203 115
BIT 6,H	203 116
BIT 6,L	203 117
BIT 6,(HL)	203 118
BIT 6,(IX+n)	221 203 n 118
BIT 6,(IY+n)	253 103 n 118
BIT 7,A	203 127
BIT 7,B	203 120
BIT 7,C	203 121
BIT 7,D	203 122
BIT 7,E	203 123
BIT 7,H	203 124
BIT 7,L	203 125
BIT 7,(HL)	203 126
BIT 7,(IX+n)	221 203 n 126
BIT 7,(IY+n)	253 203 n 126
CALL ADRESSE	205 n n
CALL C,ADRESSE	220 n n
CALL M,ADRESSE	252 n n
CALL NC,ADRESSE	212 n n
CALL NZ,ADRESSE	196 n n
CALL P,ADRESSE	244 n n
CALL PE,ADRESSE	236 n n
CALL PO,ADRESSE	228 n n
CALL Z,ADRESSE	204 n n
CCF	63
CP A	191
CP B	184
CP C	185
CP D	186
CP E	187
CP H	188
CP L	189
CP n	254 n
CP (HL)	190
CP (IX+n)	221 190 n
CP (IY+n)	253 190 n
CPD	237 169
CPDR	237 185
CPI	237 161
CPIR	237 177
CPL	47
DAA	39

ASSEMBLER KODE SORTERET ALFABETISK

DEC A	61
DEC B	5
DEC C	13
DEC D	77
DEC E	29
DEC H	37
DEC L	45
DEC (HL)	53
DEC (IX+n)	221 53 n
DEC (IY+n)	253 53 n
DEC BC	11
DEC DE	27
DEC HL	43
DEC IX	221 43
DEC IY	253 43
DEC SP	59
DI	243
DJNZ,n	16 n
EI	251
EX AF,A´F´	8
EX DE,HL	235
EX (SP),HL	227
EX (SP),IX	221 227
EX (SP),IY	253 227
EXX	217
HALT	118
IM 0	237 70
IM 1	237 86
IM 2	237 94
IN A,(C)	237 120
IN A,port	219 n
IN B,(C)	237 64
IN C,(C)	237 72
IN D,(C)	237 80
IN E,(C)	237 88
IN H,(C)	237 96
IN L,(C)	237 104
INC A	60
INC B	4
INC C	12
INC D	20
INC E	28
INC H	36
INC L	44
INC (HL)	52
INC (IX+n)	221 52 n
INC (IY+n)	253 52 n
INC BC	3
INC DE	19
INC HL	35
INC IX	221 35
INC IY	253 35
INC SP	51
IND	237 170
INDR	237 186
INI	237 162
INIR	237 178
JP ADRESSE	195 n n
JP C,ADRESSE	218 n n
JP M,ADRESSE	250 n n
JP NC,ADRESSE	210 n n
JP NZ,ADRESSE	194 n n
JP P,ADRESSE	242 n n
JP PE,ADRESSE	234 n n
JP PO,ADRESSE	226 n n
JP Z,ADRESSE	202 n n
JP (HL)	233
JP (IX)	221 233
JP (IY)	253 233
JR C,n	56 n
JR n	24 n
JR NC,n	48 n
JR NZ,n	32 n
JR Z,n	40 n

ASSEMBLER KODE SORTERET ALFABETISK

LD A,A	127
LD A,B	120
LD A,C	121
LD A,D	122
LD A,E	123
LD A,H	124
LD A,I	237 87
LD A,L	125
LD A,n	62 n
LD A,R	237 95
LD A,(ADRESSE)	58 nn
LD A,(BC)	10
LD A,(DE)	26
LD A,(HL)	126
LD A,(IX+n)	221 126 n
LD A,(IY+n)	253 126 n
LD B,A	71
LD B,B	64
LD B,C	65
LD B,D	66
LD B,E	67
LD B,H	68
LD B,L	69
LD B,n	6 n
LD B,(HL)	70
LD B,(IX+n)	221 70 n
LD B,(IY+n)	253 70 n
LD BC,nn	1 nn
LD BC,(ADRESSE)	237 75 nn
LD C,A	79
LD C,B	72
LD C,C	73
LD C,D	74
LD C,E	75
LD C,H	76
LD C,L	77
LD C,n	14 n
LD C,(HL)	78
LD C,(IX+n)	221 78 n
LD C,(IY+n)	253 78 n
LD D,A	87
LD D,B	80
LD D,C	81
LD D,D	82
LD D,E	83
LD D,H	84
LD D,L	85
LD D,n	22 n
LD D,(HL)	86
LD D,(IX+n)	221 86 n
LD D,(IY+n)	253 86 n
LD DE,nn	17 n n
LD DE,(ADRESSE)	237 91 n n
LD E,A	95
LD E,B	88
LD E,C	89
LD E,D	90
LD E,E	91
LD E,H	92
LD E,L	93
LD E,n	30 n
LD E,(HL)	94
LD E,(IX+n)	221 94 n
LD E,(IY+n)	253 94 n
LD H,A	103
LD H,B	96
LD H,C	97
LD H,D	98
LD H,E	99
LD H,H	100
LD H,L	101
LD H,n	38 n
LD H,(HL)	102
LD H,(IX+n)	221 102 n
LD H,(IY+n)	253 102 n
LD HL,nn	33 n n
LD HL,(ADRESSE)	237 107 n n
LD HL,(ADRESSE)	42 n n
LD I,A	237 71
LD IX,nn	221 33 n n
LD IX,(ADRESSE)	221 42 n n

ASSEMBLER KODE SORTERET ALFABETISK

LD IY,n n	253 33 n n
LD IY,(ADRESSE)	253 42 n n
LD L,A	111
LD L,B	104
LD L,C	105
LD L,D	106
LD L,E	107
LD L,H	108
LD L,L	109
LD L,n	46 n
LD L,(HL)	110
LD L,(IX+n)	221 110 n
LD L,(IY+n)	253 110 n
LD R,A	237 79
LD SP,HL	249
LD SP,IX	221 249
LD SP,IY	253 249
LD SP,n n	49 n n
LD SP,(ADRESSE)	237 123 n n
LD (ADRESSE),A	50 n n
LD (ADRESSE),BC	237 67 n n
LD (ADRESSE),DE	237 83 n n
LD (ADRESSE),HL	237 99 n n
LD (ADRESSE),HL	34 n n
LD (ADRESSE),IX	221 34 n n
LD (ADRESSE),IY	253 34 n n
LD (ADRESSE),SP	237 115 n n
LD (BC),A	2
LD (DE),A	18
LD (HL),A	119
LD (HL),B	112
LD (HL),C	113
LD (HL),D	114
LD (HL),E	115
LD (HL),H	116
LD (HL),L	117
LD (HL),n	54 n
LD (IX+n),A	221 119 n
LD (IX+n),B	221 112 n
LD (IX+n),C	221 113 n
LD (IX+n),D	221 114 n
LD (IX+n),E	221 115 n
LD (IX+n),H	221 116 n
LD (IX+n),L	221 117 n
LD (IX+n),n	221 54 n n
LD (IY+n),A	253 119 n
LD (IY+n),B	253 112 n
LD (IY+n),C	253 113 n
LD (IY+n),D	253 114 n
LD (IY+n),E	253 115 n
LD (IY+n),H	253 116 n
LD (IY+n),L	253 117 n
LD (IY+n),n	253 54 n n
LDD	237 168
LDDR	237 184
LDI	237 160
LDIR	237 176
NEG	237 68
NOP	0
OR A	183
OR B	176
OR C	177
OR D	178
OR E	179
OR H	180
OR L	181
OR n	246 n
OR (HL)	182
OR (IX+n)	221 182 n
OR (IY+n)	253 182 n
OTDR	237 187
OTIR	237 179

ASSEMBLER KODE SORTERET ALFABETISK

OUTD	237 171
OUTI	237 163
OUT port,A	211 n
OUT (C),A	237 121
OUT (C),B	237 65
OUT (C),C	237 73
OUT (C),D	237 81
OUT (C),E	237 89
OUT (C),H	237 97
OUT (C),L	237 105
POP AF	241
POP BC	193
POP DE	209
POP HL	225
POP IX	221 225
POP IY	253 225
PUSH AF	245
PUSH BC	197
PUSH DE	213
PUSH HL	229
PUSH IX	221 225
PUSH IY	253 225
RES 0,A	203 135
RES 0,B	203 128
RES 0,C	203 129
RES 0,D	203 130
RES 0,E	203 131
RES 0,H	203 132
RES 0,L	203 133
RES 0,(HL)	203 134
RES 0,(IX+n)	221 203 n 134
RES 0,(IY+n)	253 203 n 134
RES 1,A	203 143
RES 1,B	203 136
RES 1,C	203 137
RES 1,D	203 138
RES 1,E	203 139
RES 1,H	203 140
RES 1,L	203 141
RES 1,(HL)	203 142
RES 1,(IX+n)	221 203 n 142
RES 1,(IY+n)	253 203 n 142
RES 2,A	203 151
RES 2,B	203 144
RES 2,C	203 145
RES 2,D	203 146
RES 2,E	203 147
RES 2,H	203 148
RES 2,L	203 149
RES 2,(HL)	203 150
RES 2,(IX+n)	221 203 n 150
RES 2,(IY+n)	253 203 n 150
RES 3,A	203 159
RES 3,B	203 152
RES 3,C	203 153
RES 3,D	203 154
RES 3,E	203 155
RES 3,H	203 156
RES 3,L	203 157
RES 3,(HL)	203 158
RES 3,(IX+n)	221 203 n 158
RES 3,(IY+n)	253 203 n 158
RES 4,A	203 167
RES 4,B	203 160
RES 4,C	203 161
RES 4,D	203 162
RES 4,E	203 163
RES 4,H	203 164
RES 4,L	203 165

ASSEMBLER KODE SORTERET ALFABETISK

RES 4,(HL)	203 166
RES 4,(IX+n)	221 203 n 166
RES 4,(IY+n)	253 203 n 166
RES 5,A	203 175
RES 5,B	203 168
RES 5,C	203 169
RES 5,D	203 170
RES 5,E	203 171
RES 5,H	203 172
RES 5,L	203 173
RES 5,(HL)	203 174
RES 5,(IX+n)	221 203 n 174
RES 5,(IY+n)	253 203 n 174
RES 6,A	203 183
RES 6,B	203 176
RES 6,C	203 177
RES 6,D	203 178
RES 6,E	203 179
RES 6,H	203 180
RES 6,L	203 181
RES 6,(HL)	203 182
RES 6,(IX+n)	221 203 n 182
RES 6,(IY+n)	253 203 n 182
RES 7,A	203 191
RES 7,B	203 184
RES 7,C	203 185
RES 7,D	203 186
RES 7,E	203 187
RES 7,H	203 188
RES 7,L	203 189
RES 7,(HL)	203 190
RES 7,(IX+n)	221 203 n 190
RES 7,(IY+n)	253 203 n 190

RET	201
RET C	216
RET M	248
RET NC	208
RET NZ	192
RET P	240
RET PE	232
RET PO	224
RET Z	200
RETI	237 77
RETN	237 69
RLA	23
RL A	203 23
RL B	203 16
RL C	203 17
RL D	203 18
RL E	203 19
RL H	203 20
RL L	203 21
RL (HL)	203 22
RL (IX+n)	221 203 n 22
RL (IY+n)	253 203 n 22
RLCA	7
RLC A	203 7
RLC B	203 0
RLC C	203 1
RLC D	203 2
RLC E	203 3
RLC H	203 4
RLC L	203 5
RLC (HL)	203 6
RLC (IX+n)	221 203 n 6
RLC (IY+n)	253 203 n 6
RLD	237 111

ASSEMBLER KODE SORTERET ALFABETISK

RRA	31
RR A	203 31
RR B	203 24
RR C	203 25
RR D	203 26
RR E	203 27
RR H	203 28
RR L	203 29
RR (HL)	203 30
RR (IX+n)	221 203 n 30
RR (IY+n)	253 203 n 30
RRCA	15
RRC A	203 15
RRC B	203 8
RRC C	203 9
RRC D	203 10
RRC E	203 11
RRC H	203 12
RRC L	203 13
RRC (HL)	203 14
RRC (IX+n)	221 203 n 14
RRC (IY+n)	253 203 n 14
RRD	237 103
RST 0	199
RST 8	207
RST 16	215
RST 24	223
RST 32	231
RST 40	239
RST 48	247
RST 56	255
SBC A,A	159
SBC A,B	152
SBC A,C	153
SBC A,D	154
SBC A,E	155
SBC A,H	156
SBC A,L	157
SBC A,n	222 n
SBC A,(HL)	158
SBC A,(IX+n)	221 158 n
SBC A,(IY+n)	253 158 n
SBC HL,BC	237 66
SBC HL,DE	237 82
SBC HL,HL	237 98
SBC HL,SP	237 114
SCF	55
SET 0,A	203 199
SET 0,B	203 192
SET 0,C	203 193
SET 0,D	203 194
SET 0,E	203 195
SET 0,H	203 196
SET 0,L	203 197
SET 0,(HL)	203 198
SET 0,(IX+n)	221 203 n 198
SET 0,(IY+n)	253 203 n 198

ASSEMBLER KODE SORTERET ALFABETISK

SET 1,A	203 207
SET 1,B	203 200
SET 1,C	203 201
SET 1,D	203 202
SET 1,E	203 203
SET 1,H	203 204
SET 1,L	203 205
SET 1,(HL)	203 206
SET 1,(IX+n)	221 203 n 206
SET 1,(IY+n)	253 203 n 206
SET 2,A	203 215
SET 2,B	203 208
SET 2,C	203 209
SET 2,D	203 210
SET 2,E	203 211
SET 2,H	203 212
SET 2,L	203 213
SET 2,(HL)	203 214
SET 2,(IX+n)	221 203 n 214
SET 2,(IY+n)	253 203 n 214
SET 3,A	203 223
SET 3,B	203 216
SET 3,C	203 217
SET 3,D	203 218
SET 3,E	203 219
SET 3,H	203 220
SET 3,L	203 221
SET 3,(HL)	203 222
SET 3,(IX+n)	221 203 n 222
SET 3,(IY+n)	253 203 n 222
SET 4,A	203 231
SET 4,B	203 224
SET 4,C	203 225
SET 4,D	203 226
SET 4,E	203 227
SET 4,H	203 228
SET 4,L	203 229
SET 4,(HL)	203 230
SET 4,(IX+n)	221 203 n 230
SET 4,(IY+n)	253 203 n 230
SET 5,A	203 239
SET 5,B	203 232
SET 5,C	203 233
SET 5,D	203 234
SET 5,E	203 235
SET 5,H	203 236
SET 5,L	203 237
SET 5,(HL)	203 238
SET 5,(IX+n)	221 203 n 238
SET 5,(IY+n)	253 203 n 238
SET 6,A	203 247
SET 6,B	203 240
SET 6,C	203 241
SET 6,D	203 242
SET 6,E	203 243
SET 6,H	203 244
SET 6,L	203 245
SET 6,(HL)	203 246
SET 6,(IX+n)	221 203 n 246
SET 6,(IY+n)	253 203 n 246
SET 7,A	203 255
SET 7,B	203 248
SET 7,C	203 249
SET 7,D	203 250
SET 7,E	203 251
SET 7,H	203 252
SET 7,L	203 253
SET 7,(HL)	203 254
SET 7,(IX+n)	221 203 n 254
SET 7,(IY+n)	253 203 n 254
SLA A	203 39
SLA B	203 32
SLA C	203 33
SLA D	203 34
SLA E	203 35
SLA H	203 36
SLA L	203 37
SLA (HL)	203 38
SLA (IX+n)	221 203 n 38
SLA (IY+n)	253 203 n 38

ASSEMBLER KODE SORTERET ALFABETISK

SRA A	203 47
SRA B	203 40
SRA C	203 41
SRA D	203 42
SRA E	203 43
SRA H	203 44
SRA L	203 45
SRA (HL)	203 46
SRA (IX+n)	221 203 n 46
SRA (IY+n)	253 203 n 46
SRL A	203 63
SRL B	203 56
SRL C	203 57
SRL D	203 58
SRL E	203 59
SRL H	203 60
SRL L	203 61
SRL (HL)	203 62
SRL (IX+n)	221 203 n 62
SRL (IY+n)	253 203 n 62
SUB A	151
SUB B	144
SUB C	145
SUB D	146
SUB E	147
SUB H	148
SUB L	149
SUB n	214 n
SUB (HL)	150
SUB (IX+n)	221 150 n
SUB (IY+n)	253 150 n
XOR A	175
XOR B	168
XOR C	169
XOR D	170
XOR E	171
XOR H	172
XOR L	173
XOR n	238
XOR (HL)	174
XOR (IX+n)	221 174 n
XOR (IY+n)	253 174 n

BEMÆRK: trods ihærdige forsøg på at korrigere evt. fejl kan der alligevel være sneget sig nogle ind i tabellen. Sammenlign derfor de to tabeller med hinanden som sidste kontrol.

DECIMAL KODE SORTERET KRONOLOGISK

0	NOP
1 n n	LD BC,n n
2	LD (BC),A
3	INC BC
4	INC B
5	DEC B
6 n	LD B,n
7	RLCA
8	EX AF,A´F´
9	ADD HL,BC
10	LD A,(BC)
11	DEC BC
12	INC C
13	DEC C
14 n	LD C,n
15	RRCA
16 n	DJNZ n
17 n n	LD DE,n n
18	LD (DE),A
19	INC DE
20	INC D
21	DEC D
22 n	LD D,n
23	RLA
24 n	JR n
25	ADD HL,DE
26	LD A,(DE)
27	DEC DE
28	INC E
29	DEC E
30 n	LD E,n
31	RRA
32 n	JR NZ,n
33 n n	LD HL,n n
34 n n	LD (ADRESSE),HL
35	INC HL
36	INC H
37	DEC H
38 n	LD H,n
39	DAA
40 n	JR Z,n
41	ADD HL,HL
42 n n	LD HL,(ADRESSE)
43	DEC HL
44	INC L
45	DEC L
46 n	LD L,n
47	CPL
48 n	JR NC,n
49 n n	LD SP,n n
50 n n	LD (ADRESSE),A
51	INC SP
52	INC (HL)
53	DEC (HL)
54 n	LD (HL),n
55	SCF
56 n	JR C,n
57	ADD HL,SP
58 n n	LD A,(ADRESSE)
59	DEC SP
60	INC A
61	DEC A
62 n	LD A,n
63	CCF
64	LD B,B
65	LD B,C
66	LD B,D
67	LD B,E
68	LD B,H
69	LD B,L
70	LD B,(HL)
71	LD B,A
72	LD C,B
73	LD C,C
74	LD C,D
75	LD C,E
76	LD C,H
77	LD C,L
78	LD C,(HL)
79	LD C,A
80	LD D,B
81	LD D,C

DECIMAL KODE SORTERET KRONOLOGISK

82	LD D,D	123	LD A,E
83	LD D,E	124	LD A,H
84	LD D,H	125	LD A,L
85	LD D,L	126	LD A,(HL)
86	LD D,(HL)	127	LD A,A
87	LD D,A	128	ADD A,B
88	LD E,B	129	ADD A,C
89	LD E,C	130	ADD A,D
90	LD E,D	131	ADD A,E
91	LD E,E	132	ADD A,H
92	LD E,H	133	ADD A,L
93	LD E,L	134	ADD A,(HL)
94	LD E,(HL)	135	ADD A,A
95	LD E,A	136	ADC A,B
96	LD H,B	137	ADC A,C
97	LD H,C	138	ADC A,D
98	LD H,D	139	ADC A,E
99	LD H,E	140	ADC A,H
100	LD H,H	141	ADC A,L
101	LD H,L	142	ADC A,(HL)
102	LD H,(HL)	143	ADC A,A
103	LD H,A	144	SUB B
104	LD L,B	145	SUB C
105	LD L,C	146	SUB D
106	LD L,D	147	SUB E
107	LD L,E	148	SUB H
108	LD L,H	149	SUB L
109	LD L,L	150	SUB (HL)
110	LD L,(HL)	151	SUB A
111	LD L,A	152	SBC A,B
112	LD (HL),B	153	SBC A,C
113	LD (HL),C	154	SBC A,D
114	LD (HL),D	155	SBC A,E
115	LD (HL),E	156	SBC A,H
116	LD (HL),H	157	SBC A,L
117	LD (HL),L	158	SBC A,(HL)
118	HALT	159	SBC A,A
119	LD (HL),A	160	AND B
120	LD A,B	161	AND C
121	LD A,C	162	AND D
122	LD A,D	163	AND E

DECIMAL KODE SORETERET KRONOLOGISK

164	AND H	205 n n	CALL ADRESSE
165	AND L	206 n	ADC A,n
166	AND (HL)	207	RST 8
167	AND A	208	RET NC
168	XOR B	209	POP DE
169	XOR C	210 n n	JP NC,ADRESSE
170	XOR D	211 n	OUT (n),A
171	XOR E	212 n n	CALL NC,ADRESSE
172	XOR H	213	PUSH DE
173	XOR L	214 n	SUB n
174	XOR (HL)	215	RST 16
175	XOR A	216	RET C
176	OR B	217	EXX
177	OR C	218 n n	JP C.ADRESSE
178	OR D	219 n	IN A,(n)
179	OR E	229 n n	CALL C,ADRESSE
180	OR H	221	(se APP. C)
181	OR L	222 n	SBC A,n
182	OR (HL)	223	RST 24
183	OR A	224	RET PO
184	CP B	225	POP HL
185	CP C	226 n n	JP PO,ADRESSE
186	CP D	227	EX (SP),HL
187	CP E	228 n n	CALL PO,ADRESSE
188	CP H	229	PUSH HL
189	CP L	230 n	AND n
190	CP (HL)	231	RST 32
191	CP A	232	RET PE
192	RET NZ	233	JP (HL)
193	POP BC	234 n n	JP PE,ADRESSE
194 n n	JP NZ,ADRESSE	235	EX DE,HL
195 n n	JP ADRESSE	236 n n	CALL PE,ADRESSE
196 n n	CALL NZ,ADRESSE	237	(se afslutn. af tabel)
197	PUSH BC	238 n	XOR n
198 n	ADD A,n	239	RST 40
199	RST 0	240	RET P
200	RET Z	241	POP AF
201	RET	242 n n	JP P,ADRESSE
202 n n	JP Z,ADRESSE	243	DI
203	(se næste side)	244 n n	CALL P,ADRESSE
204 n n	CALL Z,ADRESSE	245	PUSH AF

DECIMAL KODE SORTERET KRONOLOGISK

246 n	OR n
247	RST 48
248	RET M
249	LD SP,HL
250 n n	JP M,ADRESSE
251	EI
252 n n	CALL M,ADRESSE
253	(se APP. C)
254 n	CP n
255	RST 56
203 0	RLC B
203 1	RLC C
203 2	RLC D
203 3	RLC E
203 4	RLC H
203 5	RLC L
203 6	RLC (HL)
203 7	RLC A
203 8	RRC B
203 9	RRC C
203 10	RRC D
203 11	RRC E
203 12	RRC H
203 13	RRC L
203 14	RRC (HL)
203 15	RRC A
203 16	RL B
203 17	RL C
203 18	RL D
203 19	RL E
203 20	RL H
203 21	RL L
203 22	RL (HL)
203 23	RL A
203 24	RR B
203 25	RR C
203 26	RR D
203 27	RR E
203 28	RR H
203 29	RR L
203 30	RR (HL)
203 31	RR A
203 32	SLA B
203 33	SLA C
203 34	SLA D
203 35	SLA E
203 36	SLA H
203 37	SLA L
203 38	SLA (HL)
203 39	SLA A
203 40	SRA B
203 41	SRA C
203 42	SRA D
203 43	SRA E
203 44	SRA H
203 45	SRA L
203 46	SRA (hl)
203 47	SRA A
203 56	SRL B
203 57	SRL C
203 58	SRL D
203 59	SRL E
203 60	SRL H
203 61	SRL L
203 62	SRL (HL)
203 63	SRL A
203 64	BIT 0,B
203 65	BIT 0,C
203 66	BIT 0,D
203 67	BIT 0,E
203 68	BIT 0,H
203 69	BIT 0,L
203 70	BIT 0,(HL)
203 71	BIT 0,A
203 72	BIT 1,B
203 73	BIT 1,C
203 74	BIT 1,D
203 75	BIT 1,E
203 76	BIT 1,H
203 77	BIT 1,L
203 78	BIT 1,(HL)
203 79	BIT 1,A

DECIMAL KODE SORTERET KRONOLOGISK

203 80	BIT 2,B	203 120	BIT 7,B
203 81	BIT 2,C	203 121	BIT 7,C
203 82	BIT 2,D	203 122	BIT 7,D
203 83	BIT 2,E	203 123	BIT 7,E
203 84	BIT 2,H	203 124	BIT 7,H
203 85	BIT 2,L	203 125	BIT 7,L
203 86	BIT 2,(HL)	203 126	BIT 7,(HL)
203 87	BIT 2,A	203 127	BIT 7,A
203 88	BIT 3,B	203 128	RES 0,B
203 89	BIT 3,C	203 129	RES 0,C
203 90	BIT 3,D	203 130	RES 0,D
203 91	BIT 3,E	203 131	RES 0,E
203 92	BIT 3,H	203 132	RES 0,H
203 93	BIT 3,L	203 133	RES 0,L
203 94	BIT 3,(HL)	203 134	RES 0,(HL)
203 95	BIT 3,A	203 135	RES 0,A
203 96	BIT 4,B	203 136	RES 1,B
203 97	BIT 4,C	203 137	RES 1,C
203 98	BIT 4,D	203 138	RES 1,D
203 99	BIT 4,E	203 139	RES 1,E
203 100	BIT 4,H	203 140	RES 1,H
203 101	BIT 4,L	203 141	RES 1,L
203 102	BIT 4,(HL)	203 142	RES 1,(HL)
203 103	BIT 4,A	203 143	RES 1,A
203 104	BIT 5,B	203 144	RES 2,B
203 105	BIT 5,C	203 145	RES 2,C
203 106	BIT 5,D	203 146	RES 2,D
203 107	BIT 5,E	203 147	RES 2,E
203 108	BIT 5,H	203 148	RES 2,H
203 109	BIT 5,L	203 149	RES 2,L
203 110	BIT 5,(HL)	203 150	RES 2,(HL)
203 111	BIT 5,A	203 151	RES 2,A
203 112	BIT 6,B	203 152	RES 3,B
203 113	BIT 6,C	203 153	RES 3,C
203 114	BIT 6,D	203 154	RES 3,D
203 115	BIT 6,E	203 155	RES 3,E
203 116	BIT 6,H	203 156	RES 3,H
203 117	BIT 6,L	203 157	RES 3,L
203 118	BIT 6,(HL)	203 158	RES 3,(HL)
203 119	BIT 6,A	203 159	RES 3,A

DECIMAL KODE SORTERET KRONOLOGISK

203 160	RES 4,B	203 201	SET 1,C
203 161	RES 4,C	203 202	SET 1,D
203 162	RES 4,D	203 203	SET 1,E
203 163	RES 4,E	203 204	SET 1,H
203 164	RES 4,H	203 205	SET 1,L
203 165	RES 4,L	203 206	SET 1,(HL)
203 166	RES 4,(HL)	203 207	SET 1,A
203 167	RES 4,A	203 208	SET 2,B
203 168	RES 5,B	203 209	SET 2,C
203 169	RES 5,C	203 210	SET 2,D
203 170	RES 5,D	203 211	SET 2,E
203 171	RES 5,E	203 212	SET 2,H
203 172	RES 5,H	203 213	SET 2,L
203 173	RES 5,L	203 214	SET 2,(HL)
203 174	RES 5,(HL)	203 215	SET 2,A
203 175	RES 5,A	203 216	SET 3,B
203 176	RES 6,B	203 217	SET 3,C
203 177	RES 6,C	203 218	SET 3,D
203 178	RES 6,D	203 219	SET 3,E
203 179	RES 6,E	203 220	SET 3,H
203 180	RES 6,H	203 221	SET 3,L
203 181	RES 6,L	203 222	SET 3,(HL)
203 182	RES 6,(HL)	203 223	SET 3,A
203 183	RES 6,A	203 224	SET 4,B
203 184	RES 7,B	203 225	SET 4,C
203 185	RES 7,C	203 226	SET 4,D
203 186	RES 7,D	203 227	SET 4,E
203 187	RES 7,E	203 228	SET 4,H
203 188	RES 7,H	203 229	SET 4,L
203 189	RES 7,L	203 230	SET 4,(HL)
203 190	RES 7,(HL)	203 231	SET 4,A
203 191	RES 7,A	203 232	SET 5,B
203 192	SET 0,B	203 233	SET 5,C
203 193	SET 0,C	203 234	SET 5,D
203 194	SET 0,D	203 235	SET 5,E
203 195	SET 0,E	203 236	SET 5,H
203 196	SET 0,H	203 237	SET 5,L
203 197	SET 0,L	203 238	SET 5,(HL)
203 198	SET 0,(HL)	203 239	SET 5,A
203 199	SET 0,A	203 240	SET 6,B
203 200	SET 1,B	203 241	SET 6,C

DECIMAL KODE SORTERET KRONOLOGISK

203 242	SET 6,D
203 243	SET 6,E
203 244	SET 6,H
203 245	SET 6,L
203 246	SET 6,(HL)
203 247	SET 6,A
203 248	SET 7,B
203 249	SET 7,C
203 250	SET 7,D
203 251	SET 7,E
203 252	SET 7,H
203 253	SET 7,L
203 254	SET 7,(HL)
203 255	SET 7,A
237 64	IN B,(C)
237 65	OUT (C),B
237 66	SBC HL,BC
237 67 n n	LD (ADRESSE),BC
237 68	NEG
237 69	RETN
237 70	IM 0
237 71	LD I,A
237 72	IN C,(C)
237 73	OUT (C),C
237 74	ADC HL,BC
237 75 n n	LD BC,(ADRESSE)
237 77	RETI
237 79	LD R,A
237 80	IN D,(C)
237 81	OUT (C),D
237 82	SBC HL,DE
237 83 n n	LD (ADRESSE),DE
237 86	IM 1
237 87	LD A,I
237 88	IN E,(C)
237 89	OUT (C),E
237 90	ADC HL,DE
237 91 n n	LD DE,(ADRESSE)
237 94	IM 2
237 95	LD A,R
237 96	IN H,(C)
237 97	OUT (C),H
237 98	SBC HL,HL
237 99 n n	LD DE,(ADRESSE)
237 103	RRD
237 104	IN L,(C)
237 105	OUT (C),L
237 106	ADC HL,HL
237 107 n n	LD HL,(ADRESSE)
237 111	RLD
237 112	IN F,(C)
237 114	SBC HL,SP
237 115 n n	LD (ADRESSE),SP
237 120	IN A,(C)
237 121	OUT (C),A
237 122	ADC HL,SP
237 123 n n	LD SP,(ADRESSE)
237 160	LDI
237 161	CPI
237 162	INI
237 163	OUTI
237 168	LDD
237 169	CPD
237 170	IND
237 171	OUTD
237 176	LDIR
237 177	CPIR
237 178	INIR
237 179	OTIR
237 184	LDDR
237 185	CPDR
237 186	INDR
237 187	OTDR

TALSYSTEMER:

Vi bruger to talsystemer her i bogen.Disse er det

1. decimal og
2. binære talsystem,

hvor det decimale talsystem er det system,der normalt kaldes titals systemet.Det er dette system vi bruger i dag.

TITALS SYSTEMET:

Vi mennesker er fra vor første tælle og regnelærdom trænet i at bruge titals eller det decimale talsystem.Men hvad betyder dette egentlig at bruge netop dette system ?

Lad os først kort se på hvordan vi egentlig opfatter tal skrevet i dette talsystem.Tag f.eks. tallet 5944.

Tallet består af 4 tal eller symboler.Vi bruger symbolerne fra 0 til 9 for at angive tal;når vi bruger det decimale system.Der er ialt 10 symboler.Vi bruger altså disse i kombinationer til at skrive tal større eller mindre end tal vi ikke umiddelbart kan skrive med de 10 grundsymboler.Heraf kommer bl.a. navnet titals systemet.

Når vi skal læse tallet sker det ved at adskille symbolerne i pladser eller positioner.Vi får således (for hel tal) at tallet længst mod højre er enere,derefter tiere,osv. eller sagt lidt anderledes,symbolet længst mod højre betegner et antal gange tallet opløftet i 0'te.Næste plads betegner et antal gange ti opløftet i 1'ste,osv. med potensen stigende med een for hver plads vi bevæger os mod venstre.D.v.s. at tallet 5944 skal læses:

$$5944 = 5*10^3 + 9*10^2 + 4*10^1 + 4*10^0$$
$$= 5*1000 + 9*100 + 4*10 + 4*1$$

Heraf ses,at den værdi et symbol har,foruden symbolets værdi (her tallene 5,9,4 og4) uden position får symbolet værdi efter hvilken position (plads) i rækkefølgen det har.Symbolet

får altså sit symbols grundværdi (0 til 9) gange med tallet 10 opløftet til en potens der afhænger af symbolets position.Potensen er stigende mod venstre,hvorved tal stående længst mod venstre er størst.

Vi kan summere alt dette i et sæt regler,som viser sig at være generelle for samtlige talsystemer:

1. Antallet af symboler i talsystemet er lig med basis,d.v.s. i titals systemet er 10 basis,hvorfor der må være ti tal.
2. Det symbol,som i 1. position har den største værdi er een mindre end basis.I titalsystemet er det tallet 9.
3. Større tal end de med symbolerne i 1. position angivne vil kunne konstrueres ved at det pågældende symbol multipliceres med basis,som er opløftet i den potens,der svarer til symbolets position.1. position er er pladsen længst til højre evt. lige til venstre for komma.

Disse regler er gode at kende,når man skal arbejde med et andet talsystem.Skulle vi f.eks. arbejde med et 8-tals system viste vi fra reglerne at tallet 8 er basis.Hermed ved vi allerede at der må være 8 symboler i 8-tals systemet.Tallet 7 må være det største grundsymbol.Skal vi skrive større tal end 7 skal dette gøres på formen: (hvor s står for et grundsymbol,d.v.s. et tal mellem 0 - 7)

$$s*8^4 + s*8^3 + s*8^2 + s*8^1 + s*8^0;$$

Du skulle nu kunne se,hvordan man læser tal,omsætter disse til 'vores' talsystem,fra et givet talsystem.

Vi skal nu prøve kræfter med det binære eller 2-tals systemet,som bruges meget (altid) i forbindelse med computere.Dette hænger sammen med at man kan bruge dette talsystems enkelthed i elektriske kredse.Her kan der være en vis spænding eller der kan være ingen spænding.Disse to situationer karakteriserer det binære talsystems to grundsymboler 1 og 0.

TO-TALS SYSTEMET ELLER DET BINÆRE TALSYSTEM

Det binære talsystem har basis lig med 2.Derfor må det største symbol være tallet 1,een mindre end basis (2).Af regel 1 fås,at der er to symboler tallene 1 og 0.

På engelsk kalder man tal for " digit " og tal i det binære talsystem for " binary digits ".Understregningen giver det kendte ord " bit ".En bit er altså bare et tal fra det binære talsystem,d.v.s. et 1 eller et 0.

Man har i bl.a. computere brugt en standart enhed af 8 bit i mange forbindelser,og har derfor givet denne enhed et særligt navn,nemlig en " byte ".En byte er altså blot et tal som består af 8 bit eller 8 tal fra det binære talsystem.Tilfældigvis er netop et tal på størrelse med en byte,den enhed som din computers memory maksimalt kan indeholde pr. adresse.

Når binære tal skal omsættes til decimale tal (vores) skal vi bruge regel 3 fra forrige afsnit.Denne regel sagde,at vi skulle tage symbolet (1 eller 0) og gange det med basis (2) som var opløftet i en potens,svarende til grundsymbolets position.Lad os prøve en gang med tallet

1 0 1 1 0 1 0 1

som vi skal have omskrevet til decimal tal.Vi skal altså tage tallet længst til venstre,finde positionen og vi skulle så kunne finde netop dette tals værdi.Dette skal gøres med samtlige tal og til sidst skal vi beregne summen af værdierne:

```
1 0 0 0  0 0 0 0 = 1*2^7 = 128
  0 0 0  0 0 0 0 = 0*2^6 =   0   (vi ser at når symbolet
    1 0  0 0 0 0 = 1*2^5 =  32    er nul er værdien 0)
      1  0 0 0 0 = 1*2^4 =  16
           1 0 0 = 1*2^2 =   4
               1 = 1*2^0 =   1

          sum            161.
```

Denne måde skal i princippet udføres hver gang vi skal omsætte et binært tal til decimal tal.Læg mærke til at potensen basis skal opløftes i,hver gang falder med een for hver plads vi rykker mod højre.

Det største tal vi kan skrive med een byte må så blive det tal,som har alle bit lig med 1,eller som det også siges,har samtlige bit " set ".Dette tal må være

1 1 1 1 1 1 1 1

$= 1*2^7 + 1*2^6 + 1*2^5 + 1*2^4 + 1*2^3 + 1*2^2 + 1*2^1 + 1*2^0$

$= 255.$

Dette betyder således at en byte nødvendigvis må være et tal mellem 0 og 255.Dette kan vi konstatere ved at " PEEK "e nogle adresser i ROM eller RAM.Heri findes med garanti ikke tal større end 255.

Skal vi konstruere støøre tal end 255 må vi altså bruge flere bytes.Hvis vi får f.eks. endnu en byte til rådighed bliver der altså ialt 16 bit.Som du kan se af ovenstående er bit nr. 7s værdi,når den er " set " lig med 128.Den første bit i den nye byte vil så have værdien $1*2^8 = 256$,når bitten er " set ".Det største tal man kan skrive med to bytes er 65535.

Skal vi omsætte tal den modsatte vej,d.v.s. fra det decimale talsystem til det binære er sagen lidt mere kompliceret.Hvis vi kun taler om en byte tal betyder dette,at vi skal finde ud af hvor stor en position den første bit,der er " set " kan have.Tager vi decimaltallet skal vi altså finde den største grundværdi,der er mindre end eller lig med decimaltallet. Er grundværdien mindre end decimaltallet skal vi gøre det samme med resten.Grundtallene er

positions nr.	7	6	5	4	3	2	1	0
grundværdi	128	64	32	16	8	4	2	1
	$1*2^7$	$1*2^6$	$1*2^5$	$1*2^4$	$1*2^3$	$1*2^2$	$1*2^1$	$1*2^0$

Lad os tage et eksempel,f.eks. decimaltallet 143.Her er det største grundtal 128,og dette betyder så at ihvert tilfælde bit nr. 7 skal være " set ".143-128=15 er resten.Hermed kan hverken grundtallene 64,32 eller 16 bruges,disse er for store. Men det kan grundtal 8.Hermed skal bit nr. 3 være " set ",mens

bittene imellem nr. 7 og 3 skal være " reset ".15-8=7 er resten. Vi skal igennem det samme igen.Grundtallet 4 kan nu gå fra,derefter 2 og til sidst 1,d.v.s. at bit nr. 2,1 og 0 skal være "set". Ialt får vi tallet 143 omskrevet til binær tal

143 = 1 0 0 0 1 1 1 1

Vi kan jo kontrollere dette med

$1*2^7 + 1*2^3 + 1*2^2 + 1*2^1 + 1*2^0 = 143.$

Efterhånden vil du lære hurtigt at omsætte tal direkte,uden brug af papir og blyant.Her i starten får du formentlig sjældent brug for de binære koder,så tag det helt roligt.

Afslutningsvis skal det nævnes,at der er et andet talsystem,som er meget brugt i forbindelse med computere.Det er det hexadecimale talsystem.På grund af de mange maskinkode hjælpemidler,som findes på markedet,som bruger hexadecimal,vil du på et eller andet tidspunkt stifte bekendtskab med dette.Der er imidlertid intet kompliceret ved dette talsystem.Bruger du de her viste regler for omregning kan du klare et givet talsystem.Det er valgt at undlade brugen af hexadecimal,idet du ganske udemærket kan klare dig uden.Det er brugt alene,fordi computerens memory konstruktion gør angivelse af forskellige adresser meget lette at huske og omregne i hexadecimal.Ellers er det blot et tidkrævende mellemled,idet du ikke kan " POKE " hexadecimaltal ind i memory, men konstant skal omsætte disse til decimale tal.

Det er måske på sin plads kort at karakterisere hexadecimal systemet.Hexa betyder 16,d.v.s. 16-tals systemet.Der er derfor 16 symboler fra 0 - 9,mens tallene 10 - 15 angives ved bogstaverne A til F.Skal vi omsætte tal fra dette system til decimal foregår dette på nøjagtig samme måde,som vi gjorde det med de binære tal,her er 16 bare basis,som skal opløftes i potens.

INDEKS-REGISTER INSTRUKTIONERNES DECIMAL KODE SORTERET KRONOLOGISK:

221 9	ADD IX,BC
221 25	ADD IX,DE
221 33 n n	LD IX,n n
221 34 n n	LD (ADRESSE),IX
221 35	INC IX
221 41	ADD IX,IX
221 42 n n	LD IX,(ADRESSE)
221 43	DEC IX
221 52 n	INC (IX+n)
221 53 n	DEC (IX+n)
221 54 n n	LD (IX+n),n
221 57	ADD IX,SP
221 70 n	LD B,(IX+n)
221 78 n	LD C,(IX+n)
221 86 n	LD D,(IX+n)
221 94 n	LD E,(IX+n)
221 102 n	LD H,(IX+n)
221 110 n	LD L,(IX+n)
221 112 n	LD (IX+n),B
221 113 n	LD (IX+n),C
221 114 n	LD (IX+n),D
221 115 n	LD (IX+n),E
221 116 n	LD (IX+n),H
221 117 n	LD (IX+n),L
221 119 n	LD (IX+n),A
221 126 n	LD A,(IX+n)
221 134 n	ADD A,(IX+n)
221 142 n	ADC A,(IX+n)
221 150 n	SUB (IX+n)
221 158 n	SBC A,(IX+n)
221 166 n	AND (IX+n)
221 174 n	XOR (IX+n)
221 182 n	OR (IX+n)
221 190 n	CP (IX+n)
221 203 n 6	RLC (IX+n)
221 203 n 14	RRC (IX+n)
221 203 n 22	RL (IX+n)
221 203 n 30	RR (IX+n)
221 203 n 38	SLA (IX+n)
221 203 n 46	SRA (IX+n)
221 203 n 62	SRL (IX+n)
221 203 n 70	BIT 0.,(IX+n)
221 203 n 78	BIT 1.,(IX+n)
221 203 n 86	BIT 2.,(IX+n)
221 203 n 94	BIT 3.,(IX+n)
221 203 n 102	BIT 4.,(IX+n)
221 203 n 110	BIT 5.,(IX+n)
221 203 n 118	BIT 6.,(IX+n)
221 203 n 126	BIT 7.,(IX+n)
221 203 n 134	RES 0.,(IX+n)
221 203 n 142	RES 1.,(IX+n)
221 203 n 150	RES 2.,(IX+n)
221 203 n 158	RES 3.,(IX+n)
221 203 n 166	RES 4.,(IX+n)
221 203 n 174	RES 5.,(IX+n)
221 203 n 182	RES 6.,(IX+n)
221 203 n 190	RES 7.,(IX+n)
221 203 n 198	SET 0.,(IX+n)
221 203 n 206	SET 1.,(IX+n)
221 203 n 214	SET 2.,(IX+n)
221 203 n 222	SET 3.,(IX+n)
221 203 n 230	SET 4.,(IX+n)
221 203 n 238	SET 5.,(IX+n)
221 203 n 246	SET 6.,(IX+n)
221 203 n 254	SET 7.,(IX+n)
221 225	POP IX
221 227	EX (SP),IX
221 229	PUSH IX
221 233	JP (IX)
221 249	LD SP,IX

Udskift 221 med 253 og samtlige instruktioner for " IY " haves.

FLAG-EFFEKT

På de næste to sider følger effekten på flagene i " F "-registret.Der er brugt 5 symboler til at illustrere flaget tilstand efter at instruktionen,som står i margen er eksekveret.Symbolerne og deres betydning er vist i figr 1. nedenfor.

Tegnet	- ? -	betyder,at flaget er afhængig af resultatet.Slå tilbage til gennemgangen af " F "-registret.
Tegnet	- ÷ -	betyder,at flaget ikke påvirkes.
Tegnet	- -	(der er intet) betyder,at effekten er vilkårlig.
Tegnet	- 0 -	betyder,som vanlig at flaget er " reset ".
Tegnet	- 1 -	betyder,at flaget er " set ".

Figur 1. Symboler forklaring til flag-effekt skema.

OVERSIGT OVER FLAG-EFFEKT

ASSEMBLER KODE	S	Z	H	P/V	N	C
LD register,tal	÷	÷	÷	÷	÷	÷
LD register,register	÷	÷	÷	÷	÷	÷
LD register,(registerpar)	÷	÷	÷	÷	÷	÷
LD (registerpar),tal	÷	÷	÷	÷	÷	÷
LD (registerpar),register	÷	÷	÷	÷	÷	÷
LD (adresse),registerpar	÷	÷	÷	÷	÷	÷
LD registerpar,tal	÷	÷	÷	÷	÷	÷
LD registerpar,registerpar	÷	÷	÷	÷	÷	÷
LD registerpar,(adresse)	÷	÷	÷	÷	÷	÷
ADD tal	?	?	?	?	0	?
ADD register	?	?	?	?	0	?
ADD (registerpar)	?	?	?	?	0	?
ADC A,tal	?	?	?	?	0	?
ADC A,register	?	?	?	?	0	?
ADC A,(registerpar)	?	?	?	?	0	?
INC register	?	?	?	?	0	÷
INC registerpar	÷	÷	÷	÷	÷	÷
SUB tal	?	?	?	?	1	?
SUB register	?	?	?	?	1	?
SUB (registerpar)	?	?	?	?	1	?
SBC A,tal	?	?	?	?	1	?
SBC A,register	?	?	?	?	1	?
SBC A,(registerpar)	?	?	?	?	1	?
DEC register	?	?	?	?	1	?
DEC registerpar	÷	÷	÷	÷	÷	÷
CP tal	?	?	?	?	1	?
CP register	?	?	?	?	1	?
CP (registerpar)	?	?	?	?	1	?

ASSEMBLER KODE	S	Z	H	P/V	N	C
AND tal	?	?	1	?	0	0
AND register	?	?	1	?	0	0
AND (registerpar)	?	?	1	?	0	0
OR tal	?	?	0	?	0	0
OR register	?	?	0	?	0	0
OR (registerpar)	?	?	0	?	0	0
XOR tal	?	?	0	?	0	0
XOR register	?	?	0	?	0	0
XOR (registerpar)	?	?	0	?	0	0
ADD registerpar,registerpar	÷	÷		÷	0	?
ADC registerpar,registerpar	?	?		?	0	?
SBC registerpar,registerpar	?	?		?	1	?
PUSH registerpar	÷	÷	÷	÷	÷	÷
POP registerpar	÷	÷	÷	÷	÷	÷
CALL adresse	÷	÷	÷	÷	÷	÷
CALL flag,adresse	÷	÷	÷	÷	÷	÷
RET	÷	÷	÷	÷	÷	÷
RET flag	÷	÷	÷	÷	÷	÷
CPI	?	?	?	?	1	÷
CPD	?	?	?	?	1	÷
CPIR	?	?	?	?	1	÷
CPDR	?	?	?	?	1	÷
LDI	÷	÷	0	?	0	÷
LDD	÷	÷	0	?	0	÷
LDIR	÷	÷	0	0	0	÷
LDDR	÷	÷	0	0	0	÷
EX registerpar,registerpar	÷	÷	÷	÷	÷	÷

ASSEMBLER KODE	S	Z	H	P/V	N	C
RLC register	?	?	?	?	?	?
RLC (registerpar)	?	?	?	?	?	?
RLCA	÷	÷	÷	÷	÷	÷
RRC register	?	?	?	?	?	?
RRC (registerpar)	?	?	?	?	?	?
RRCA	÷	÷	÷	÷	÷	÷
RL register	?	?	?	?	?	?
RL (registerpar)	?	?	?	?	?	?
RLA	÷	÷	÷	÷	÷	÷
RR register	?	?	?	?	?	?
RR (registerpar)	?	?	?	?	?	?
RRA	÷	÷	÷	÷	÷	÷
SLA register	?	?	?	?	?	?
SLA (registerpar)	?	?	?	?	?	?
SRA register	?	?	?	?	?	?
SRA (registerpar)	?	?	?	?	?	?
SRL register	0	?	?	?	?	?
SRL (registerpar)	0	?	?	?	?	?

OVERSIGT OVER OPERATIONS-TIDER

ASSEMBLER KODE	TID	ANTAL BYTES
LD register,register	4	1
LD register,tal	7	2
LD register,(registerpar)	7	1
LD A,(adresse)	13	3
LD registerpar,tal	10	3/4
LD registerpar,(adresse)	16/20	3/4
PUSH registerpar	11	1
POP registerpar	10	1
ADD register	4	1
ADD tal	7	2
ADC A,register	4	1
ADC A,tal	7	2
INC register	4	1
INC(registerpar)	11	1/3
JP adresse	10	3
JR flag bytes	12	2
JR bytes	12	2
CALL adresse	12/7	3
RET	10	1
RLA	4	1

Ovenviste liste er ikke udtømmende.Du kan finde samtlige instruktions operationstid,antal bytes m.m. i bogen " Z-80 CPU INSTRUKTION SET ",som er skrevet af Alan Tootill.Bogen er svært tilegnelig,grundet dens tekniske krav til læseren.Absolut ikke en bog for en begynder.Senere vil du dog kunne drage stor nytte af den,når du har fået indlært alle instruktioner og dermed nået et højere programmerings niveau.I øvrigt er bogen temmelig dyr. Der er udgivet en lignende af ZILOG fabrikken selv.Osse dyr,men måske bedre rent indlærings mæssigt.

- Hurtigere BASIC-programmer
- Bedre udnyttelse af AMSTRADs operativsystem
- Udførelse af to programmer samtidigt
- Bedre udnyttelse af lageret

er blot nogle af de fordele du vil få ved at lære at programmere din AMSTRAD-datamat i maskinkode. Dette er netop målet med denne bog.

Uden forkundskaber til maskinkode vil læseren i takt med læsningen af bogen lære alle instruktionerne, der kan anvendes i maskinkodeprogrammering.

Herudover fås en væsentligt bedre indsigt i hvordan datamaten arbejder i udførelsen af BASIC-programmer. Således vil flere rutiner i operativsystemet blive omtalt og vist anvendelsen af, hvilket giver såvel BASIC- som maskinkodeprogrammer større fleksibilitet, eksempelvis anvendelse af interruptsystem, styring af kassettebåndoptageren, Monitoren etc.

COPENHAGEN BOOK CENTRE ApS

Frederiksberg Bogtrykkeri 850016

ISBN 87-88739-02-3

AMSTRAD MASKINKODE